Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1993-03

The design and implementation of an object-oriented interface for the multimodel/multilingual database system

Moore, John William.; Karlidere, Turgay

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/39874

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A267 413

DTIC
ELECTE
JUL 27 1993
S E D

THESIS

DESIGN AND IMPLEMENTATION OF AN OBJECT-ORIENTED INTERFACE FOR THE MULTI-MODEL/MULTILINGUAL DATABASE SYSTEM

by

John William Moore
and
Turgay Karlidere

March 1993

Thesis Advisor: D. K. Hsiao

Approved for public release; distribution is unlimited.

93-16843

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
THE DESIGN AND IMPLEMENTATION OF AN OBJECT-ORIENTED INTERFACE FOR THE (Continued)

12. PERSONAL AUTHOR(S)
Moore, John William and Karlidere, Turgay

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	March 1993	82

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Object-oriented data model, Database schema, Object-oriented data language, Multimodel/Multilingual Database System, (Continued)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Database designs in today's information-intensive environment, challenge the database-system user to adhere to strict and somewhat archaic means, i.e., traditional data models and their data languages, of expressing their database applications. In light of these requirements, the user must purchase the new database system that supports the latest data model and its data language. We design and implement a comprehensive data-model-and-data-language interface which is a simple and yet effective alternative to the costly and cumbersome standard method of purchasing or developing a new database system. Our solution is two-fold. First, we use the concept of a data-model-and-data-language interface to an existing database system. This not only eliminates the costs associated with building a separate, stand-alone database system to support each new data model and its language, but also allows for resource consolidation and data duplication elimination. Second, using the data-model-and-data-language interface concept, we design and implement an object-oriented-data-model-and-data-language interface for the multimodel/multilingual database system.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Prof. David K. Hsiao	22b. TELEPHONE (Include Area Code): (408) 656-2253 / 22c. OFFICE SYMBOL: CS/37

11. MULTIMODEL/MULTILINGUAL DATABASE SYSTEM.

18. Object-oriented databases, inheritance, encapsulation, reusability, object instance, class hierarchy, generalization and specialization, and Multibackend database supercomputer.

Approved for public release; distribution is unlimited

THE DESIGN AND IMPLEMENTATION OF AN OBJECT-ORIENTED INTERFACE FOR THE MULTIMODEL/MULTILINGUAL DATABASE SYSTEM

by

John William Moore
Lieutenant, United States Navy
B.S., New Hampshire College, 1983
and
Turgay Karlidere
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1987

DTIC QUALITY INSPECTED 5

Accesion For	
NTIS CRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

Authors:

[redacted]

John William Moore

[redacted]

Turgay Karlidere

Approved By:

[redacted]

David K. Hsiao, Thesis Advisor

[redacted]

Thomas Wu, Second Reader

[redacted]

Cdr G. Hughes, Acting Chairman,
Department of Computer Science

ABSTRACT

Database design in today's information-intensive environment, challenge the database-system user to adhere to strict and somewhat archaic means, i.e., traditional data models and their data languages, of expressing their database applications. In light of these requirements, the user must purchase the new database system that supports the latest data model and its data language. We design and implement a comprehensive data-model-and-data-language interface which is a simple and yet effective alternative to the costly and cumbersome standard method of purchasing or developing a new database system. Our solution is two-fold. First, we use the concept of a data-model-and-data-language interface to an existing database system. This not only eliminates the costs associated with building a separate, stand-alone database system to support each new data model and its language, but also allows for resource consolidation and data duplication elimination. Second, using the data-model-and-data-language interface concept, we design and implement an object-oriented-data-model-and-data-language interface for the multimodel/multilingual database system.

TABLE OF CONTENTS

I. AN INTRODUCTION 1
  A. WHAT IS THE OBJECT-ORIENTED DESIGN? 7
  B. THE ORGANIZATION OF THIS THESIS 9
II. THE SYSTEM ORGANIZATION 10
  A. THE MULTIBACKEND DATABASE SUPERCOMPUTER 10
  B. THE MULTIMODEL/MULTILINGUAL DATABASE SYSTEM 12
III. THE ATTRIBUTE-BASED AND OBJECT-ORIENTED DATA MODELS 15
  A. THE ATTRIBUTE-BASED DATA MODEL 15
    1. A Conceptual View 15
    2. The Attribute Based Data Language (ABDL) 16
  B. THE OBJECT-ORIENTED DATA MODEL 17
    1. A Conceptual View 18
    2. Object-Oriented Features and Constructs 19
    3. The Object-Oriented Schema 23
    4. The Object-Oriented Data Language 26
  C. TWO MAPPING METHODS 26
    1. Mapping the Object-Oriented Data Model 27
    2. Mapping the Object-Oriented Query 31
IV. BASIC IMPLEMENTAION ISSUES 37
  A. OUR IMPLEMENTATION STRATEGY 37
  B. DATA STRUCTURES 37
    1. Data Structures Shared by all Users 38
    2. Data Structures Specific to each User 41
  C. THE DESCRIPTION OF THE MODEL ALGORITHMS 43
    1. Language Interface Layer (LIL) 43
    2. Kernel Mapping System (KMS) 45
    3. Kernel Controller (KC) 46
    4. Kernel Formatting System (KFS) 47
V. OTHER IMPLEMENTATION ISSUES 48
  A. SCHEMA MODIFICATIONS 48
  B. USER-DEFINED OPERATIONS 50
  C. OBJECTID MAINTENANCE 50
VI. OUR CONCLUSION 52
  A. LIMITATIONS 53
  B. FUTURE RESEARCH 54
APPENDIX A 55
APPENDIX B 57
APPENDIX C 68
LIST OF REFERENCES 72
INITIAL DISTRIBUTION LIST 74

I. AN INTRODUCTION

Database design in today's information-intensive environment is becoming increasingly complex. This complexity consists of not only the increasing size of corporate/ government databases, but also the timely access and production of results and answers. Thus, the manipulation of a database, in order to obtain results (which is commonly referred to as database management, or popularly referred to as data engineering) becomes important. This importance indicates that the data storage is no longer a major concern. Another factor becoming less of a concern is the particular computer on which the database system resides.

Given no restrictions on the data storage and the support computer, the proliferation of stand-alone database systems leads itself to an unnecessary amount of duplications in databases and in database transaction results. These database-system proliferation and data duplication, in turn, lead to expensive operations with respect to data upkeep and software maintenance. With various database applications in this proliferation and duplication, the use of different data models and their languages for these applications becomes necessary. As newer and more semantically-rich data models and their languages (in terms of its constructs and features) are developed, the introduction of newer database systems will accelerate the proliferation and duplication. Finally, the cost of upgrading the existing database system to a newer database system, based on a new data model and language, can also become prohibitive. We need a solution to overcome the database-system proliferation and database duplication.

Our solution is twofold. First, we incorporate the idea of an interface approach to an existing database system. This interface approach is used for separate and distinct data models and their data languages (each of which is of course specific to an application) to access a single-system database. Second, we design and implement an object-oriented data

model and data language interface. Our object-oriented interface is the motivation for this thesis.

Both elements of our solution, when combined, eliminate the needless and costly database-system proliferation and data duplication. We implement this solution on an existing system designed to support the database interface. The system is the Multimodel/ Multilingual Database System at the Laboratory for Database Systems Research in the Naval Postgraduate School. See Figure 1.

Our approach to accommodating new data models and their languages, and therefore, new database applications, is a working and effective means to eliminate all the current database and software proliferation and duplication. Also, this approach provides users with new data models and their languages for their new applications. We argue that it is not necessary to build an entire new database system for a new data model and its language. Instead, we merely need to build a new data-model-and-data-language interface on an existing database system, thereby minimizing the proliferation of stand-alone, heterogenous database systems [HsDK92].

To support our database-system-interface approach we identify four contributing factors. These factors are (1) Resource Consolidation; (2) Data Sharing; (3) Trend towards a Single-System Environment; and (4) Reduction of development and transition costs for stand-alone database system.

The first two factors, resource consolidation and data sharing, go hand-in-hand to support our solution. Resource consolidation is the combining of multiple entities performing the same functions with respect to a database management system. Consolidating the resources that make up a database system reduces redundancy and the overall database system size and cost.

The second element, data sharing, is a direct result of resource consolidation. Since all the resources have been consolidated, there is a natural progression towards providing users the means to share consolidated resources such as common data. Along with the reduction in data duplication, maintaining the data becomes an easier task.

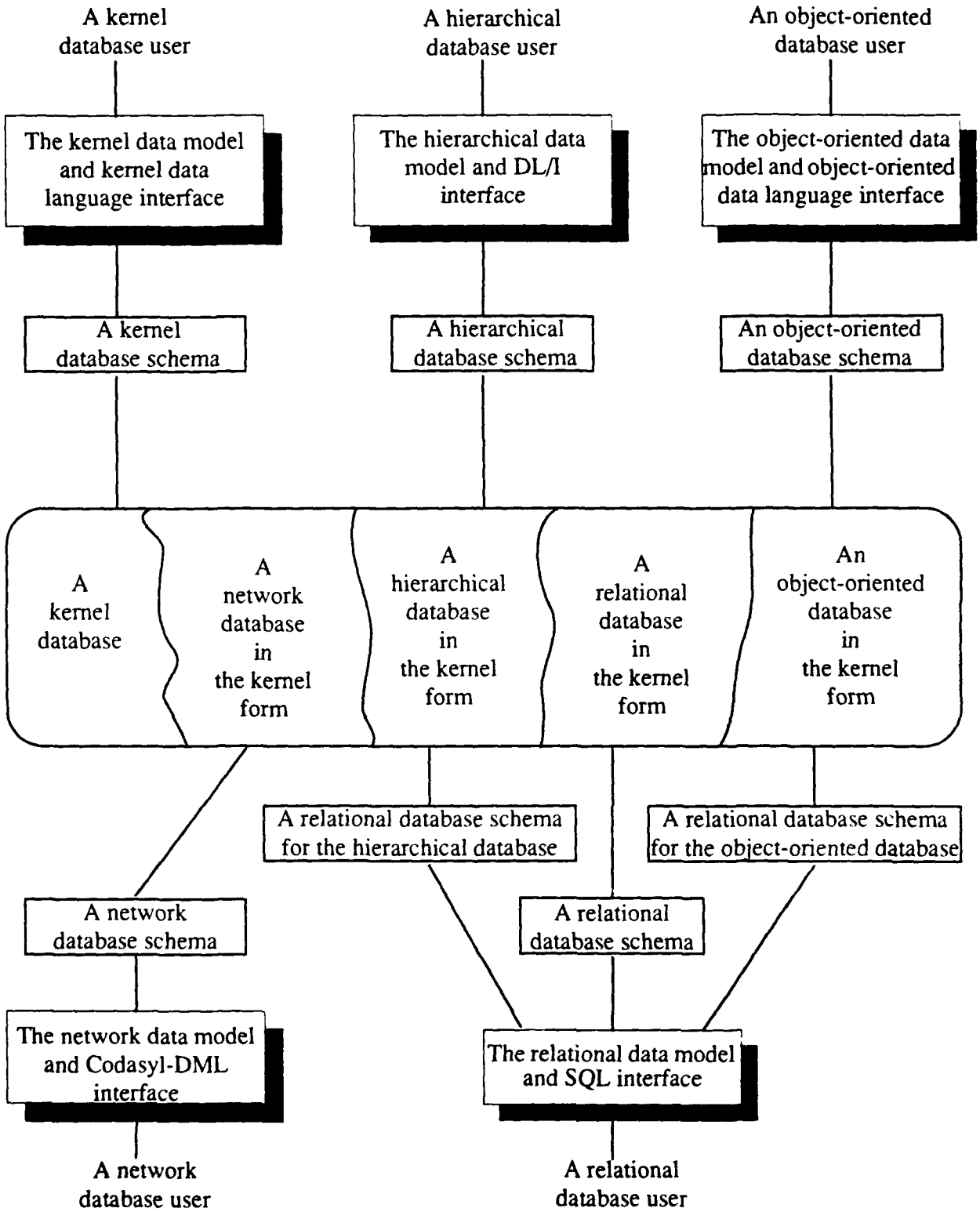


Figure 1. The Multimodel/Multilingual Database System and Cross-Model Accessing Capability

Other benefits of resource consolidation and data sharing lead into the third interface-approach supporting factor, the trend towards single-system environments. Our approach to multiple data models and data languages is a single-system one, despite its diversity in data models and data languages. This single-system approach also eliminates the problems of the standard multi-system environment where separate and heterogenous database systems are required to share data and consolidate resources [HsDK92]. In our single-system environment, each user may access the common database through different database schemas based on different data models with their corresponding database transactions versed in different data languages, respectively. Furthermore, the consolidation of all data management resources in a single-system becomes an easier task.

The above three factors, resource consolidation, data sharing, and single-system environments, when combined, help produce the fourth factor, lower development and transition costs associated with an interface-approach. By lower development costs, we refer to the minimal amount of time required to design and implement an interface to an existing database system. This minimum amount of time, compared to more conventional means, is far less than the time it takes to design and implement a stand-alone database system. Not only is the time difference less for our interface, but new and comprehensive data models and their data languages can be incorporated into the existing database system quickly. Thus, with new data models and languages, a user has more flexibility to better model his/her applications in a rapidly changing information-intensive environment.

In the traditional approach to building a new database system, once a new system has been installed, the transition to that new system is costly. However, with our interface-approach this transition is quicker and allows better continuity throughout the transition phase. With our interface approach, the transition only involves learning a new data model and its language. There are no new system features to learn. There are no new data format requirements. There is only the requirement to learn new syntax associated with a data

language in order to access the database. In affect, the only transition cost for our approach is the time it takes to learn a new data model and language.

Again these four factors enrich a users ability to effectively model their application. Combining resources and sharing data lead to a natural progression of a single-system environment. As a result of a single-system environment, the costs to develop and transition to a new data model and language are significantly less than that for a separate, stand-alone database system. The database-system-interface approach is an effective means to reap the benefits of new data models and languages quickly. Our aim now is to show how we use this approach in the design and implementation of an object-oriented interface for the multimodel/multilingual database system.

To further support our reason to design and implement an object-oriented interface vice building a stand-alone system, there are two considerations. The first consideration takes full advantage of the object-oriented data model's rich semantics with respect to its constructs and features. The object-oriented properties combine the object inheritance with the class encapsulation to form a coherent and easy-to-understand concept, the class hierarchy. The second consideration deals with the object-oriented data model's ability to add new properties and constructs to further enhance its appeal and modeling capabilities. This capability to add new properties and constructs, to be elaborated on in later chapters, will greatly enhance this new data model's flexibility to adjust to the changing environment.

To fully realize the benefits of accessing an existing database system through a new data-model-and-data-language interface, we continue our supporting arguments by introducing our object-oriented interface. Our method, via this object-oriented interface to an existing database system, alleviates those expensive developmental and transitional costs mentioned above. As we have shown in our method, the time involved from initial design to the demonstrable implementation is shorter than the design and implementation effort in a stand-alone system.

Given current database application requirements, standard data models and their data languages can only solve current application requirements. On the other hand, some new applications need new or different data models and their languages for the database management tasks.

In addition to the introduction of object-oriented database management, our motivation for the design and implementation of an object-oriented interface is to promote a new and emerging alternative to database system design. The alternative, through means of developing an object-oriented interface vice the cumbersome construction of an entire and homogeneous database system, is more effective in terms of the development time and production cost. These two factors, in addition to those mentioned above, with regards to total database-system construction, may encourage potential users of making an investment into this new technology. Further, there is no proliferation of new databases and new system software on the existing hardware.

Our design goal is to develop an object-oriented interface for the multimodel/ multilingual database system. This system currently supports relational, hierarchical, and network interfaces as well as its cross-model accessing capabilities. These and other capabilities of the multimodel/multilingual database system have been documented in [Hsia91]. We will not elaborate on them in this thesis. However, in the following paragraphs we describe briefly its architecture in order to show how various modules of the object-oriented interface are fitted into the total multimodel/multilingual database system architecture referring to Figure 1.

The first part of the multimodel/multilingual database system is considered the frontend or the user-interface portion of a two-part system. The second part of the database system is the multibackend database supercomputer [HsiD92]. The multibackend database supercomputer consists of a controller and multiple database processors and their database stores which can range in numbers from one to many. However our emphasis is on the design and implementation of the object-oriented interface to the multimodel/multilingual database system, not on multibackend database supercomputer.

A final note for our motivation, is to prove the viability for the development and implementation of our interface approach. Also, to introduce a new data model and its new data language quickly, instead of building a full-size, stand-alone database system from scratch. This interface alternative to the standard and somewhat archaic way of making use of a new data-model-and-language technology will introduce this new technology more rapidly. In this thesis, we focus on the object-oriented data-model-and-language interface. Through this interface, the object-oriented constructs and notions that are better suited to meet the ever-increasing reliance on object-oriented applications becomes a reality.

A. WHAT IS THE OBJECT-ORIENTED DESIGN?

Object-oriented design has been around for several years, yet its appeal has caught on only recently [Booc91]. Despite its appeal, there does not exist any standard object-oriented data model and its standard data language. Nevertheless, the wonderful constructs, that enable the database administrator to produce an almost exact representation of his/her application via the object-oriented data model, do exist.

These object-oriented constructs and structures consisting of, but not limited to, inheritance, encapsulation, and data abstraction, are formed into a class hierarchy. The concept of a class hierarchy, through the use of inheritance, produces and easy-to-understand relationship among the various classes within the hierarchy. In addition to the class hierarchy, there are its instances. However, given that these class hierarchical-instances are new and may be difficult to appreciate when they are first introduced, we attempt to explain this concept in the following chapters.

Object-oriented design allows the user/database administrator to view all entities of an application as objects. These objects are the primary impetus behind the success of an object-oriented design. Through the object-oriented design, examples of objects might include real-world entities, such as cars, trucks, chairs, employees, bank accounts, or even kitchen sinks. The list of objects within any object-oriented design is left to the user's imagination. With these objects in mind the database designer may now combine the

similar objects into classes of objects. For instance, the object, chair, may be put into a class of similar-featured chairs. These similar-featured chairs have four legs, a seat, and a back rest. However, some chairs may recline, roll on wheels or have arm rests. The same similar features (four legs, a seat, and a back rest) still exist but now the chair class may have a subclass of chairs. Each chair subclass of the superclass of chairs will inherit all the features of that superclass but can also be considered a class of its own. Thus, each class of chairs has both its uniqueness and inheritance preserved.

The above chair superclass/subclass example can be carried over to a vehicle-type superclass/subclass. This vehicle example, adapted and modified from [Kim90], involves a class named Vehicle. The Vehicle class has several attributes which characterize all the Vehicles. These Vehicles are themselves objects and relate to the other class by way of pointers. The term pointer is represented in the object-oriented design as an objectid of an object in the other class (described in Chapter III). The objectid is located within the superclass, Vehicle, as well as the Base class or Root class. The superclass Vehicle may have one or more subclasses that inherit its attributes. The subclasses may also have attribute peculiar to that subclass. The class Vehicle example will be elaborated on in later sections.

Additional applications that have been benefited from the object-oriented design are computer-aided design, computer aided software engineering, office information systems management and modeling and simulation. Each application has its own particular requirement for expressing its relationships among the objects and the classes. Yet each application also benefits from the object-oriented design's class concept, inheritance, encapsulation, and reusability. Consider the modeling and simulation application. The various components that go into making a prototype may be thought of as objects which may be grouped together into a class. When configuring different variations of a prototypical system, an object may be reused without degrading or affecting the other parts of the prototype. The term object reusability expresses the notion that an object, once defined, may be used again in another application with little modifications to the original

definition. A more in-depth explanation of the object reusability issue may be found in [Booc91]; however, it is beyond the scope of our thesis.

Our intent throughout this thesis is to incorporate the many features of the object-oriented design into an object-oriented interface for the multimodel/multilingual database system. Also, the user can create an object-oriented database for the user's application. Further, the user can utilized those object-oriented features implemented in our interface for writing their transactions. These transactions will be executed by the multimodel/ multilingual database system for data management operations, the multibackend database supercomputer is already configured with standard integrity, persistence, and data security features. Therefore, our interface does not have to address these system issues. Instead, our interface focuses on the object-oriented database management.

B. THE ORGANIZATION OF THIS THESIS

In the remaining chapters of this thesis we first describe the multibackend database supercomputer and the multimodel/multilingual database system in Chapter II. In Chapter III, we introduce the attribute-based and object-oriented data models. Each data model has a brief overview with examples and also details their respective data languages. However, the description on the object-oriented data model goes into a greater detail. The detail includes the object-oriented features, notions and constructs. While detailing the object-oriented data model, references to the Vehicle database are made. In Chapter IV, we cover the implementation issues. In Chapter V, we investigate other implementation issues. In Chapter VI, we summarize our accomplishments and limitations.

II. THE SYSTEM ORGANIZATION

Before describing the object-oriented interface, it is important to become familiar with the system organization upon which the interface will be implemented. The basic system organization consists of the multibackend database supercomputer, the multimodel/multilingual database system, and the attribute-based data model/language as described by Hsiao and Kemal in [Hsia89]. Even though our research was strictly on the multimodel/multilingual database system and the attribute-based data model/language, the basic system organization helps us to gain a familiarity with the overall system architecture.

A. THE MULTIBACKEND DATABASE SUPERCOMPUTER

As described in [Elma89], a simplified database system environment consists of users accessing a database system through application queries. The queries interact with the database management system software which in turn accesses the meta-data (data about the stored data) and the actual stored data. This approach, albeit effective, can be further improved by using multiple backend computers connected in parallel. Each backend computer has its own disk system controller, meta disk and a stored data disk. All the backend computers are further controlled by a backend controller which supervises the execution of user transactions. Hence, we term the architecture, the multibackend database supercomputer. See Figure 2.

The multibackend database supercomputer exhibits two capabilities: (1) Given a user query, there is a response-time reduction for the query inversely proportional to a given number of backend computers; and (2) If the number of backends increase proportionally to that of the database capacity increase, there would be no change in transaction response-time. In essence, if a user wants to increase his/her database capacity and yet maintain a reasonable transaction response-time then he/she need only to reconfigure the system with as many additional backend computers as required.

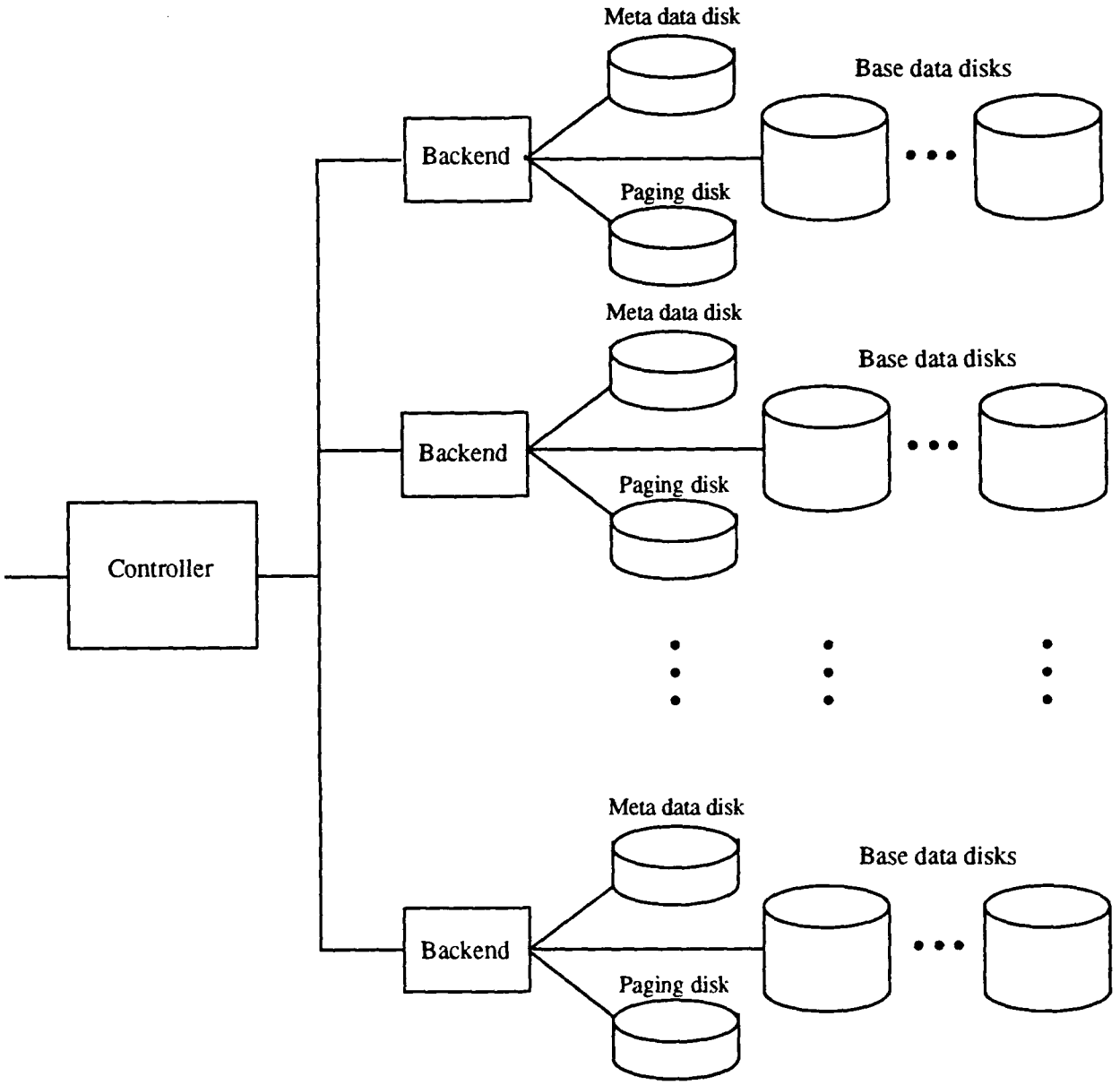


Figure 2 - The Multibackend Database Supercomputer

B. THE MULTIMODEL/MULTILINGUAL DATABASE SYSTEM

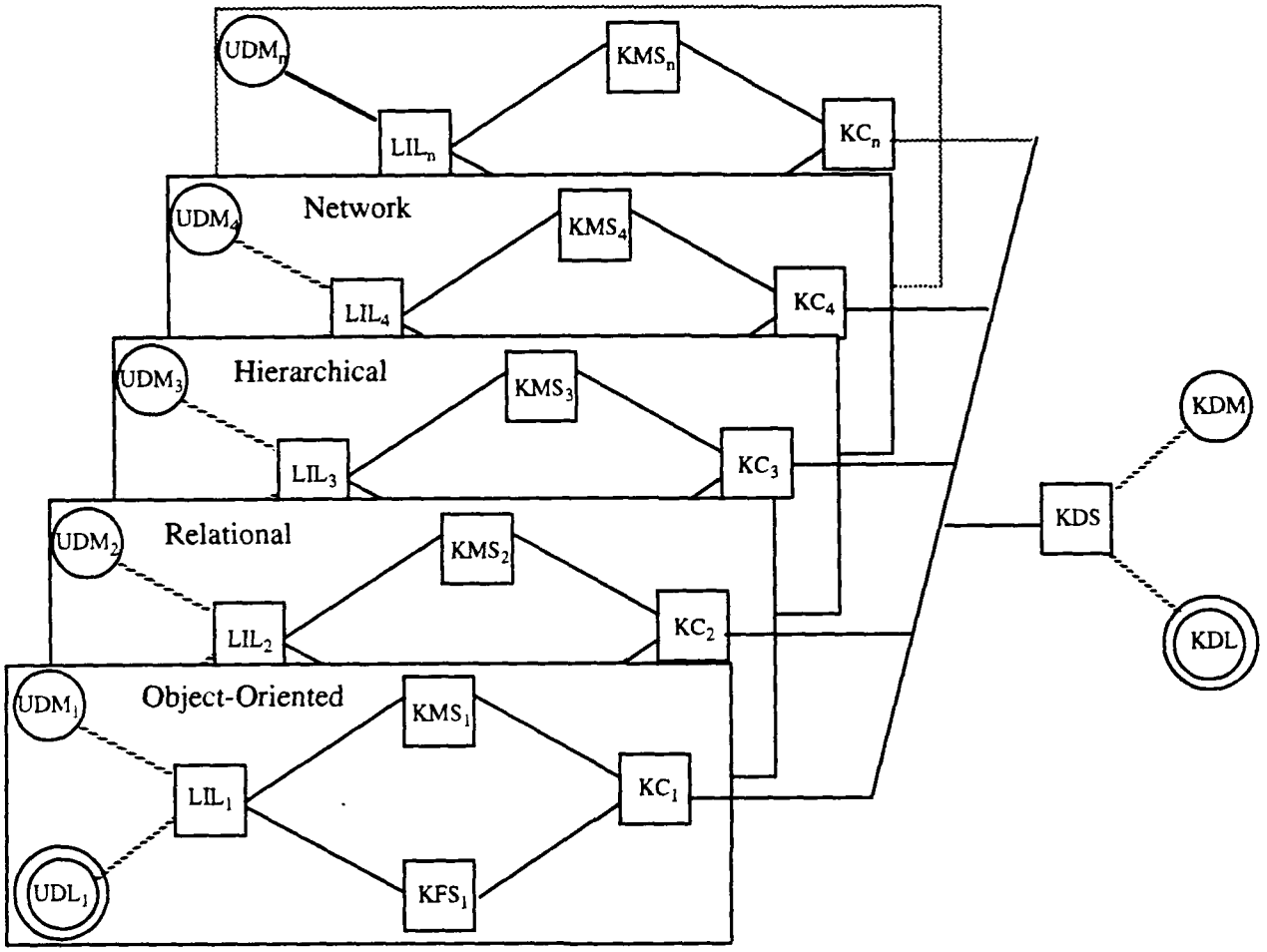
The above description of the multibackend database supercomputer provided a general idea of the hardware aspect of the system. We now describe the software aspect. In Figure 3, the multimodel/multilingual database system is shown. It provides a pictorial representation of the system modules with their respective control flows [Hsia91]. The four main modules, the language interface layer (LIL), the kernel mapping system (KMS), the kernel controller (KC) and the kernel formatting system (KFS) depict the core system for each separate user interface. The kernel database system (KDS) represents the go-between system of the kernel data model/language (KDM/L) and the user data model/language (UDM/L). These components make up the multimodel portion of the multimodel/multilingual database interface and are described individually below.

LIL routes the user's transaction written in UDM/L to the KMS. KMS has two functions. The first identifies whether or not the user is creating a new database. If the user is creating a new database, it transforms the UDM-database definition to the KDM-database definition. This is known as the data-model transformation. Once the KDM-database definition has been established, KMS sends it to KC which in turn routes the KDM-database definition to KDS. KDS then issues the appropriate commands to the multibackend database supercomputer controller where a new database is created in the KDM form. KDS then notifies KC that a new database has been created in the UDM form and data may now be entered as well as subsequent transactions against the database.

The second function of KDS is the processing of the UDL transaction. In this processing, KMS translates the UDL transaction into an equivalent KDL transaction. This is know as the data-language translation. KMS routes the KDL transaction to KC which then sends the KDL transaction to KDS for execution. KC's primary role, in this case, is to oversee the KDL transaction execution.

The KDL transaction is executed in KDS. Any answer or response is sent to KC which routes them to KFS for the KDM-to-UDM transformation. Once the transformation is complete, KFS routes it to LIL for the final relay to the user in the user's data model/language form.

Again, the overall language-interface structure consists of the LIL, KMS, KC, and KFS modules, allowing the multimodel/multilingual database system to incorporate different data



UDM - User Data Model
UDL - User Data Language
LIL - Language Interface Layer
KMS - Kernel Mapping System
KC - Kernel Controller
KDS - Kernel Database System
KDM - Kernel Data Model
KDL - Kernel Data Language
KFS - Kernel Formating System

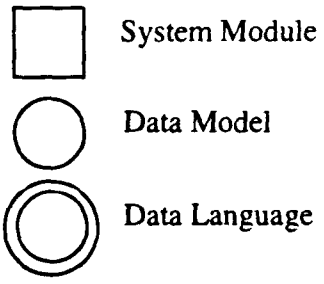


Figure 3 . The Multi-model/Multi-lingual Database System

models and data languages as described in above. KDS represents the kernel database system unique to the multibackend database supercomputer and the multimodel/multilingual database system. Each user may create/access a database using his or her data model/language but the system stores only one set of data which is in the kernel-data-model form, i.e., in the attribute-based data model.[Hsia91]

III. THE ATTRIBUTE-BASED AND OBJECT-ORIENTED DATA MODELS

In the following sections, we present the reader with an overview of the attribute-based data model and the object-oriented data model. We begin with the attribute-based data model, then end with the object-oriented data model. Our overview of the attribute-based database model, although is brief, will provide a basic understanding of the kernel data model of the multimodel/multilingual database system. The object-oriented data model is discussed in some greater detail for two reasons. The first reason is that the object-oriented data model is a new concept and requires some greater detail to describe its constructs and notions. The second reason is that our elaboration allows us to illustrate a user's application in a clear and concise manner. Consequently, they will enable the user to obtain a realistic conceptual view of their application environment. Once the conceptual view is obtained, it will be transformed into its logical representation of the modeled application. From this logical representation, the physical implementation is more easily fulfilled.

A. THE ATTRIBUTE-BASED DATA MODEL

The attribute-based data model as described by [HsDK92] is a powerful yet simple data model. This data model has advanced capabilities that allow many other data models to be mapped into it without losing any information during the data-model transformation process. It is without question the best solution as the kernel data model of the multimodel/multilingual database system.

1. A Conceptual View

In defining the attribute-based data model, our discussion begins with the database. The database is made up of records. Each piece of data in a record is in the form of an attribute-value. The attribute-value pair, being a member of the Cartesian product of attributes and values, consists of the attribute name and the value of that attribute. An example would look like the following, <WEIGHT, 2200>. The attribute-value pair example defines Weight as the attribute and 2200 as the value for that attribute. A record including the Weight attribute is shown next:.

(<TEMP, Vehicle>,<TYPE, US_Made>,<WEIGHT, 2200>,<COLOR, Silver>, {text})

The words enclosed in the angled brackets,<,>, represent attribute-value pairs, for short keywords. In particular, certain attribute-value pair is termed the directory keyword (i.e., TEMP) and its value (i.e., Vehicle). The curly brackets, {,}, enclose the record body. The entire record is completely enclosed within the parentheses.

As a requirement, each record must have at least a keyword, e.g., a *TEMP* attribute along with its corresponding value. A directory of keywords is created in the database system. The keyword directory helps to identify a record whose keyword is kept in the directory. In this case, *TYPE* is an additional directory keyword, whereas *WEIGHT* and *COLOR* might be non-directory keywords which are not kept in the directory. However, the eventual identification of a database record can be either by a directory or non-directory keyword. The advantage to using a directory keyword is a much smaller search space and hence a quicker response-time.

2. The Attribute Based Data Language (ABDL)

The multimodel aspect of the multimodel/multilingual database system is based on the attribute-based data model and has been described. Now, the multilingual portion is described next. It consists of multiple user data languages (UDL's), and a kernel data language (KDL). The attribute-based data language (ABDL) is used as the KDL in the multimodel/multilingual database system. The other user data languages, or UDLs, supported by this system are the Relational/SQL [Roll84], Hierarchical/DL-I [Weis84], and Network/CODASYL[Wort85] model/data languages. However, our thesis concentrates on the object-oriented data model/language interface.

ABDL is the data language for the multi-backend database supercomputer and the multimodel/multilingual database system. We begin our discussion by describing its record structure and operations. The ABDL database records can be identified by keyword predicates. A keyword predicate consists of a three-tuple which has the form: an attribute, a relational operator (=, ≠, >, <, ≥, ≤), and an attribute value, e.g., ENGINE_SIZE = 1600. Database records can be identified by multiple keyword predicates or a conjunction of keyword predicates.

Each expression enclosed in parentheses may be combined with another expression by the conjunctive operator, *and*. This conjunct may then be connected with another conjunct by the disjunctive operative, *or*. The entire expression, called the disjunctive of conjuncts, results in an

ADBL statement. ADBL supports four basic database operations, RETRIEVE, INSERT, UPDATE, and DELETE. Each is described below.

The RETRIEVE request is used to access the database without altering its contents. The RETRIEVE format is

RETRIEVE (query) [Target-List] [BY Attribute].

The reserved word, RETRIEVE, indicates a retrieve-type operation. The *query* specifies which records are to be retrieved. The *Target-List* identifies the attributes to output and may also include one of the following aggregate operations: AVG, COUNT, SUM, MIN, MAX. The *BY* clause is optional and will output the desired attribute values by a specified attribute.

The INSERT request inserts a new record. The example below shows the attributes, WEIGHT and ENGINE-SIZE, being inserted into the Vehicle database.:

INSERT (<TEMP, Vehicle>, <WEIGHT, 1950>, <ENGINE-SIZE, 1600>).

The UPDATE request modifies records in the database. Its syntax combines a query part with a modifier part. The query part specifies which records to modify and the modifier part indicates how to modify the record.

UPDATE (TEMP, Forgnco) (NumAutoProd = NumAutoProd + 50).

The DELETE request deletes one or more database records. The following DELETE statement illustrates how *Trucks* weighing over *2000* pounds are deleted from the database.:

DELETE ((TEMP = Truck) and (TONNAGE > 2000)).

B. THE OBJECT-ORIENTED DATA MODEL

Now that the attribute-based data model and data language have been discussed, we now focus on the object-oriented data model. Our object-oriented data model is used to provide a conceptual representation of real world objects. Along with these real-world objects are the constraints on them and their relationships to other objects [Elma89]. These objects are realized through the object-oriented data model's features. These features help the user to view of the user's environment. We will discuss the following four aspects of these object-oriented features: (1) A

Conceptual View, (2) Features and Constructs, (3) Database Schema, and (4) Object-Oriented Data Language.

1. A Conceptual View

The object-oriented data model mentioned in [Hsia92], subsumes the capabilities of other classical data models. It is characterized as a data model rich with features. These features, when combined with the notion of the object, become a powerful modeling tool.

The basic element of the object-oriented data model is the object. This object can be any entity in an application. Once the application's objects are identified, they are combined into classes of similar-objects. The object-oriented class has a class definition and is comprised of the following:

Class Name
Objectid
Class Relationship: Superclass/Subclass/Component_of
Attributes
Actions (Methods).

The *Class Name* is the name assigned to a particular class of similar objects. The *Objectid* is used to differentiate one object from the other within the database of objects. The *Class Relationship* defines the relationships of the classes of objects. There are three class relationships: superclass, subclass, and component_of. The *Superclass* relationship is a class from which all subclasses derive. It is also known as the generalization of its subclasses. The *Subclass* relationship, known as a specialization of the superclass, represents a class of objects that inherit (i.e., have) the superclass' properties (i.e., attributes and actions). It, too, has unique properties. The *Component_of* relationship identifies a particular attribute in a class that is a pointer to another class. Essentially, the "result" of the *Component_of* relationship is a class within a class. However, to illustrate the *Component_of* relationship, a separate class is identified. This class is "pointed to" by the *Component_of* attribute value. The *Attributes* are the variables which take up the specific values of a class. They describe uniquely a set of values in each object within the class. The *Actions*, also referred to as methods, are allowed operations on individuals objects of the class. Our *actions* implemented in the multimodel/multilingual database system are RETRIEVE and INSERT only. Each *action* is described in section III.C.2.

These classes of similar objects are then formed into a class hierarchy. Figure 4, the Vehicle Class Hierarchy, is an example of a class hierarchy/composition adapted from [Kim90]. Figure 5, shows an alternate conceptual view of the Vehicle class hierarchy with its style adapted from [Tsud91]. The class hierarchy incorporates class relationships as well as their respective constraints. Once the class hierarchy has been established, data may be entered into the database. The combination of the above object-oriented database components result in the object-oriented schema. However, before we discuss the schema, it is important to understand the features and constructs that define it. These features and constructs are described next.

2. Object-Oriented Features and Constructs

The features and constructs we have identified fall into two categories. The first category has two supporting concepts and deals with the class hierarchy. In order to compose this hierarchy, the designer may use the concepts of class generalization and specialization by way of the class inheritance. The second category deals with the object modules. Each object module has three features to its design: encapsulation, reusability, and object instance. These three features, combined with the class hierarchy features, distinguish the object-oriented data model from all previous data models. We elaborate on each feature below.

The class hierarchy has two distinct features, class generalization and specialization, and their inheritance. Class generalization and specialization are used to construct the class hierarchy. Referring to Figure 4, the generalized class is the VEHICLE class. It is a generalization of the VEHICLE class' subclasses, TRUCK and AUTOMOBILE. All common properties of the subclasses are maintained by the generalized class. In this case, the common attributes, *Objectid*, *Model* and *Manufacturer*, are stored in the superclass VEHICLE. The superclass VEHICLE maintains these attributes and their values within its structure. In figure 4, we also show the top class, or *Root* class. The *Root* class maintains the four class operations: INSERT, RETRIEVE, UPDATE, and DELETE.However, only the INSERT and RETRIEVE operations have been implemented. Both are discussed in section III.C.2. For our purposes, all subclass operations within the class hierarchy originate in the *Root* class.

On the other hand, the subclasses TRUCK and AUTOMOBILE are specializations of the superclass VEHICLE. These specialized classes have their common attributes in the generalized

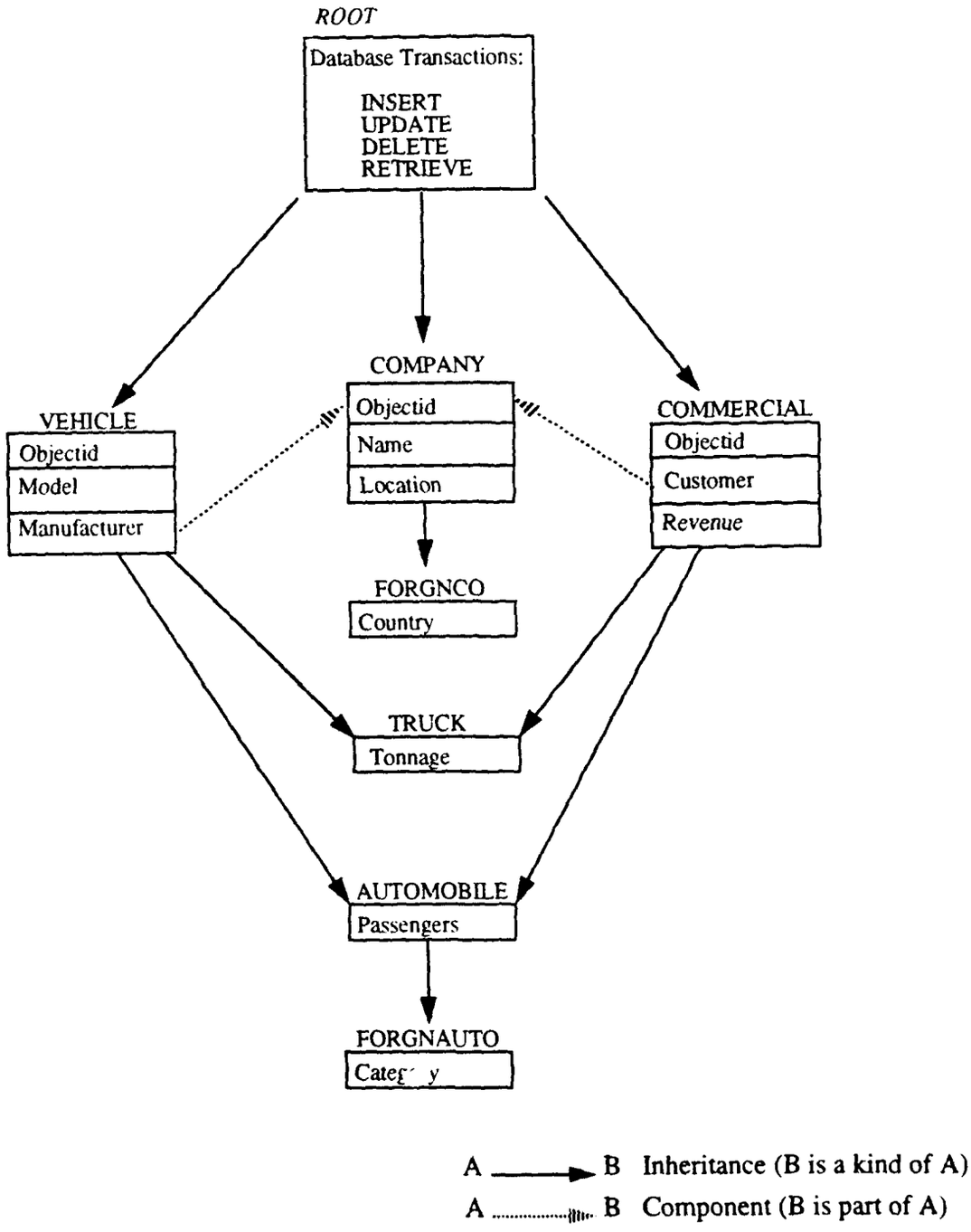


Figure 4. The VEHICLES Class Hierarchy

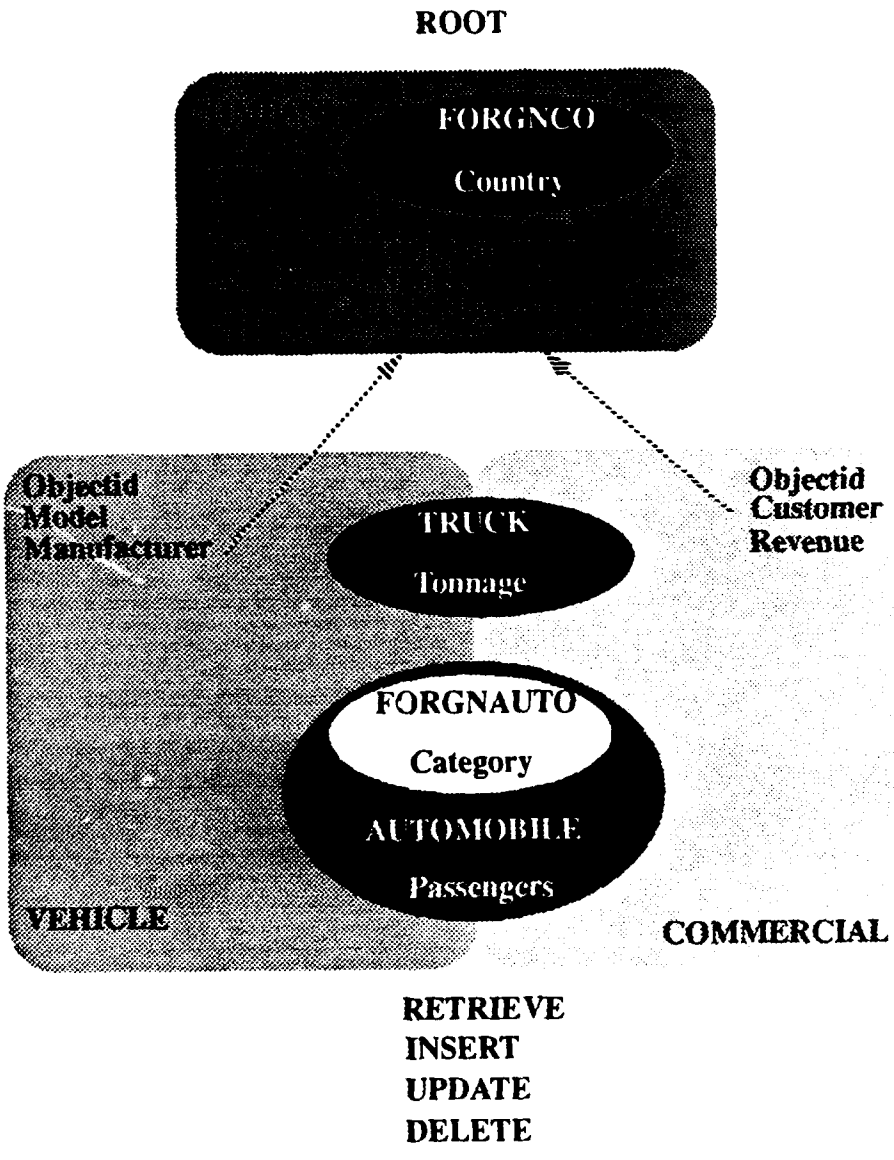


Figure 5. The Alternate View of the VEHICLES Class Hierarchy

class. They also have unique attributes and values within their own class. To further illustrate, the subclass FORGNAUTO, is a specialization of the generalized class, AUTOMOBILE. The class AUTOMOBILE is not only the generalized class of its subclass FORGNAUTO, but it is also the specialized class for the VEHICLE superclass.

Class inheritance, or simply, inheritance, is the linking element that helps define the class hierarchical composition. Through the inheritance, a specialized class inherits its superclass' properties, i.e., the attributes and actions. In our Figure 4 example, the subclass FORGNAUTO inherits *Passengers* from AUTOMOBILE. Another way to express this inheritance is to say FORGNAUTO is a-kind-of AUTOMOBILE. The a-kind-of relationship is an easy way to express class similarities, i.e., their inherited properties.

The a-kind-of relationship is an example of single-class inheritance, or single inheritance. By single inheritance we mean an object's properties are inherited from a single superclass. There is another aspect of inheritance which is called multiple inheritance. This form of a-kind-of relationship incorporates the notion of single inheritance with multiple inheritance only there are two superclasses from which a subclass inherits. An example of multiple inheritance from the VEHICLE and COMMERCIAL superclasses consists of common subclasses, TRUCK and AUTOMOBILE. See Figure 4 again. The combined inherited properties from TRUCK and AUTOMOBILE are *Objectid*, *Model*, and *Manufacturer* from the VEHICLE superclass and *Customer* and *Revenue* for the COMMERCIAL superclass. For clarity, both subclasses inherit the same Objectid. Also, since AUTOMOBILE inherits from two superclasses, these collective properties are passed on to the subclass FORGNAUTO.

The second category, dealing with object modules, has two features. Of these features, encapsulation is discussed first. Encapsulation is the principle that confines each class into a module [Schl90]. Encapsulating modules incorporates another principle. This principle is called data abstraction. Data abstraction combines an object's attributes with the operations allowed on those attributes. Hence, the object is considered not only in terms of certain attributes, but also in the ways to be operated on.

Since the module is encapsulated, accessing it is achieved through well-defined external interface. The interface is handled by a module operation which answers an outside-module

request. For example, we may request to retrieve the number of passengers a car can hold for a certain model. The VEHICLE superclass separates all cars by the requested model. It then requests an interface with the subclass AUTOMOBILE. Once this interface is established, the AUTOMOBILE class issues a command to retrieve the number of passengers for the specified model. The result of both class' retrieve commands provide the user with his/her requested information.

The encapsulation principle represents the method by which we access modules. These modules represent the way on object's data, or object instance, are identified. An object's instance consists of the domain values of all attributes for each object. For example, an instance of the subclass FORGNAUTO would include all of its superclass' properties as well.

FORGNAUTO Class

Objectid	Model	Manufacturer	Customer	Revenue	Passengers	Category
15	Prelude	2	5	250	6	Sports

The attribute values for each class' attributes, within the class hierarchy, make up the object-instance structure.

Each feature presented, highlight the uniqueness of the object-oriented data model. Our design incorporates these features but with some modifications. These few modifications will be described in Chapter IV. However, none of the modifications hinder our ability to fully realize the interface-approach. Two other aspects of our approach are the object-oriented database schema and our object-oriented data language. Each is discussed below.

3. The Object-Oriented Schema

The object-oriented schema is the description, or template, of how the data are configured in a database. All object-oriented characteristics within the database are in the schema form. A general object-oriented database schema is shown in Figure 6.

To begin, each class module encapsulates the class name and properties with its associated relationship in the class hierarchy. This relationship is shown in the form of SUPCLASS (superclass) and SUBCLASS (subclass). A class may have multiple superclasses as well as multiple subclasses. For additional classes in the hierarchy, the class structure is the same.

```
CLASS name_1
        SUPCLASS     name_1p1
                       •
                       •
                       •
        SUPCLASS     name_1pn
        SUBCLASS     name_1b1
                       •
                       •
                       •
        SUBCLASS     name_1bn
            attribute_name_1a1   type*
                       •
                       •
                       •
            attribute_name_1an   type
@
                       •
                       •
                       •
@
CLASS name_i
        SUPCLASS     name_ip1
                       •
                       •
                       •
        SUPCLASS     name_ipn
        SUBCLASS     name_ib1
                       •
                       •
                       •
        SUBCLASS     name_ibn
            attribute_name_ia1   type
                       •
                       •
                       •
            attribute_name_ian   type
$
```

(*) **type** is either a class name representing the component_of relationship, an INTEGER, a FLOAT, or a single character.

Figure 6. The General Format of an Object-Oriented Database Schema Definition File

```
CLASS VEHICLE
        SUBCLASS    AUTOMOBILE
        SUBCLASS    TRUCK
            OBJECTID            INTEGER
            MODEL               CHAR  10
            MANUFACTURER        COMPANY
@
CLASS COMMERCIAL
        SUBCLASS    AUTOMOBILE
        SUBCLASS    TRUCK
            OBJECTID            INTEGER
            CUSTOMER            COMPANY
            REVENUE             INTEGER
@
CLASS AUTOMOBILE
        SUPCLASS    VEHICLE
        SUPCLASS    COMMERCIAL
        SUBCLASS    FORNAUTO
            PASSENGERS          INTEGER
@
CLASS TRUCK
        SUPCLASS    VEHICLE
        SUPCLASS    COMMERCIAL
            TONNAGE             INTEGER
@
CLASS COMPANY
        SUBCLASS    FORNCO
            OBJECTID            INTEGER
            NAME                CHAR  10
            LOCATION            CHAR  10
@
CLASS FORGNAUTO
        SUPCLASS    AUTOMOBILE
            CATEGORY            CHAR  10
@
CLASS FORGNCO
        SUPCLASS    COMPANY
            COUNTRY             CHAR  10
$
```

Figure 7. The Object-Oriented Database Schema for the VEHICLES Database

To illustrate our schema, Figure 7 shows the actual VEHICLE Class Hierarchy. Each class within the hierarchy and its associated properties are mapped in the general schema format. For instance, AUTOMOBILE is the name of one class. Its relationship identifier within the class hierarchy is denoted by the SUPCLASS and SUBCLASS keywords. The AUTOMOBILE class also has one attribute, *Passenger*. Thus, the class AUTOMOBILE with one attribute has two superclasses, VEHICLE and COMMERCIAL. It also has one subclass, FORGNAUTO. Since each class is mapped according to the general schema format, the user has an accessible means of maintaining his/her application.

4. The Object-Oriented Data Language

There are two considerations that went into our object-oriented data language development. First, the object-oriented data language must be easy to use and simple to understand. The user should be able to write transactions with ease without getting overwhelmed by the syntax. We use a few basic terms to handle this consideration. Second, the user must allow the object-oriented data language to be mapped into the attribute-based data language efficiently.

We have developed our object-oriented data language to handle the above considerations. The actual object-oriented-to-attribute-based data language mapping is covered in section III.C.

C. TWO MAPPING METHODS

The two mapping methods we describe deal with mapping the data model and the database query. The first method, mapping the data model, closely resembles that of the IRIS system as described in [Hugh91]. Our approach, like that in the IRIS system, is to require that each object is responsible for its unique properties, but *passes* on only the object id. The object id, in this case, is similar in concept of a pointer. The pointer, or object id, for each object, uniquely identifies that object. That uniqueness is passed on, or propagated, throughout the class hierarchy. However, the actual storage of our class hierarchy along with an object's objectid and respective properties are handled by the MDBS. However, our concern is with the mapping of the objects to their attribute-based equivalents.

The second mapping method is mapping the database query. There are two queries which we will detail. The first is the Insert statement. The second query is the RETRIEVE statement. Each query-mapping is described below.

1. Mapping the Object-Oriented Data Model

The goal of mapping the object-oriented data model to the attribute-based model is to allow the system to represent the user's modeled application. In order for this model representation to occur, the user must have a clear picture of the application components. Once a clear picture is obtained, a flat file representation of the objects and classes may be generated. This is also referred to as translating the application's conceptual view to its logical view. The conceptual view is the class-hierarchical representation of the application. This representation is then transcribed into a general object-oriented schema format similar to that shown in Figure 7. These two steps help the user to transform an application into an object-oriented conceptual view. From the conceptual view, mapping it to the logical view occurs next.

We want to map the object-oriented data model to the attribute-based data model. The mapping process is handled in two phases. The first phase maps the object-oriented data model's class definition, which includes the class name and its attributes. This mapping is on an one-to-one basis with the attribute-based data model's record structure. The second phase incorporates the object-oriented class relationships within the attribute-based data model. The first phase is not difficult. However, phase two presents some unique design considerations.

During the first phase, the class-name attribute is mapped to an attribute-based equivalent-type attribute, namely TEMP. The attribute names of the object-oriented data model are used for the attribute-based data model's attributes. Referring again to Figure 7, we show an actual mapping involving the VEHICLE class. The class VEHICLE has the following definition::

Class Name : VEHICLE
Supclass : BASE CLASS
Subclass : TRUCK
Subclass : AUTOMOBILE
$Attribute_1$: OBJECTID
$Attribute_2$: MODEL
$Attribute_3$: MANUFACTURER

The *Class Name* attribute is mapped to the attribute-based TEMP attribute. While the attribute names for *Attributes*, are mapped to OBJECTID, MODEL, and MANUFACTURER attribute-based attribute names, respectively. The *Supclass* and *Subclass* relationships will be

described later. The above object-oriented-data-model-to-attribute-based-data-model mapping is shown below.

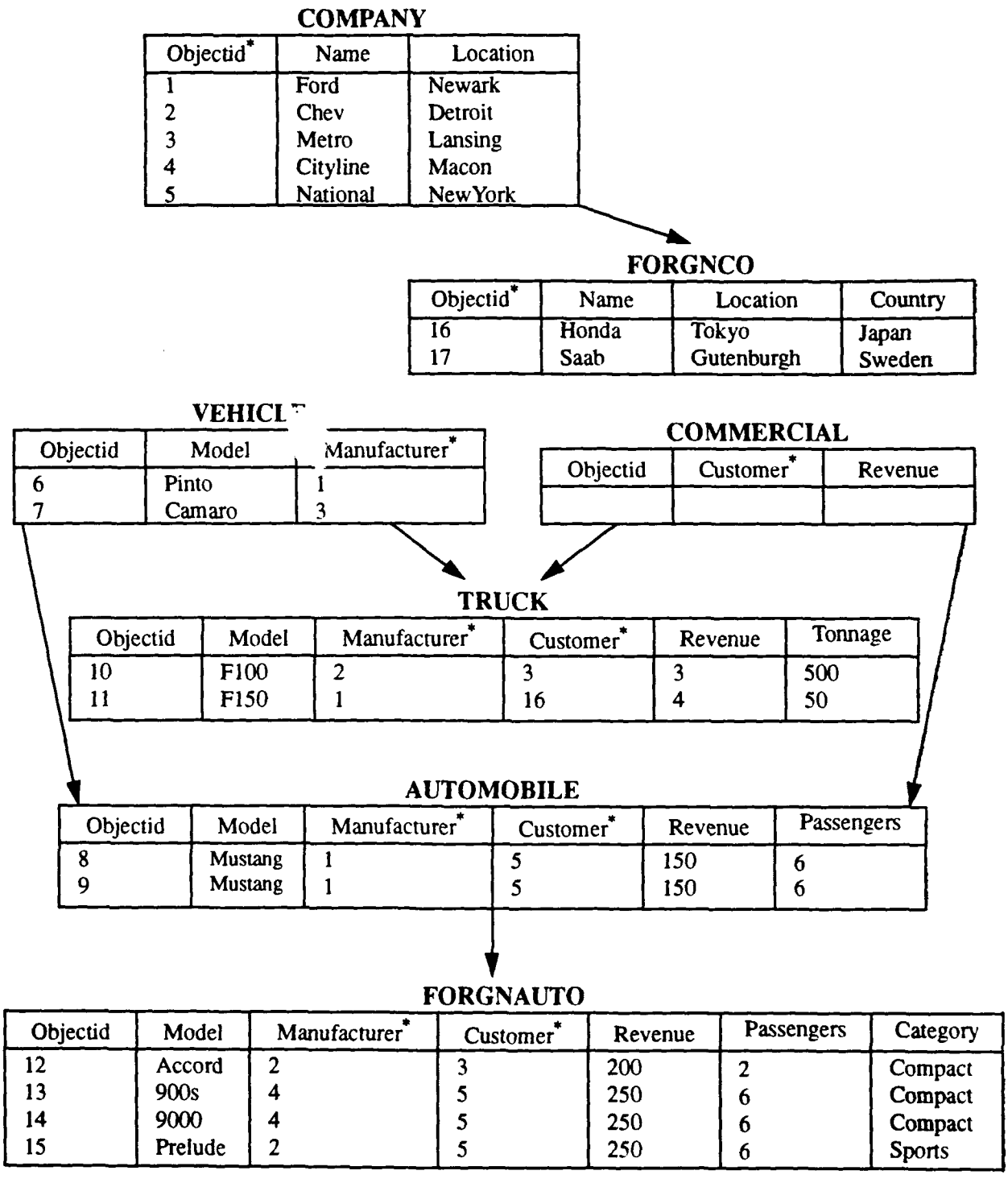
TEMP	OBJECTID	MODEL	MANUFACTURER

For each class in the VEHICLE Class Hierarchy, the mapping is similar. Now that the attribute-based template has been built, mapping the class relationships are next.

The class relationship mapping has two possible strategies. The first strategy allows for no duplication. The second strategy is an alternative method to the first strategy which does have some duplication. We choose the latter strategy and believe that the duplication which does occur is insignificant when compared to the overall data management performance benefits. Figure 8 shows the first strategy with no duplication. Figure 9 illustrates our implemented strategy. We begin our discussion with the first strategy in order to identify the design weakness. We then follow with supporting arguments the weaknesses for the second strategy which overcome the weaknesses of the first strategy. [Hugh91]

The first mapping strategy places each class instance as far down the class hierarchy as possible. In other words, there are no superclasses that store class-common properties, i.e., attributes and actions. Each class stores its own properties. The benefit to this strategy is in performing instance updates, i.e., updating individual object instances. Since each class with a superclass does not rely on the superclass to store its common properties, there is no need to propagate any update changes throughout the hierarchy. However, a problem occurs when attempting to retrieve all class instances. When retrieving all instances it becomes difficult to identify those instances requested are either specific to the class or should the subclasses be included as well. This problem of retrieving is not a problem for the second mapping strategy.

The main difference between the first strategy and our strategy of choice is we store class-common properties as high in the class hierarchy as possible. This not only allows the inheritance principle to be realized, but also assists the data management responsibilities. The inheritance principle is upheld by keeping class-unique properties within that particular class. Also, the properties common to all classes within the class hierarchy are maintained in the superclass. The



(*) Component relationship

Figure 8. The Storage of Attribute Values in the class instances to which the Values directly belong

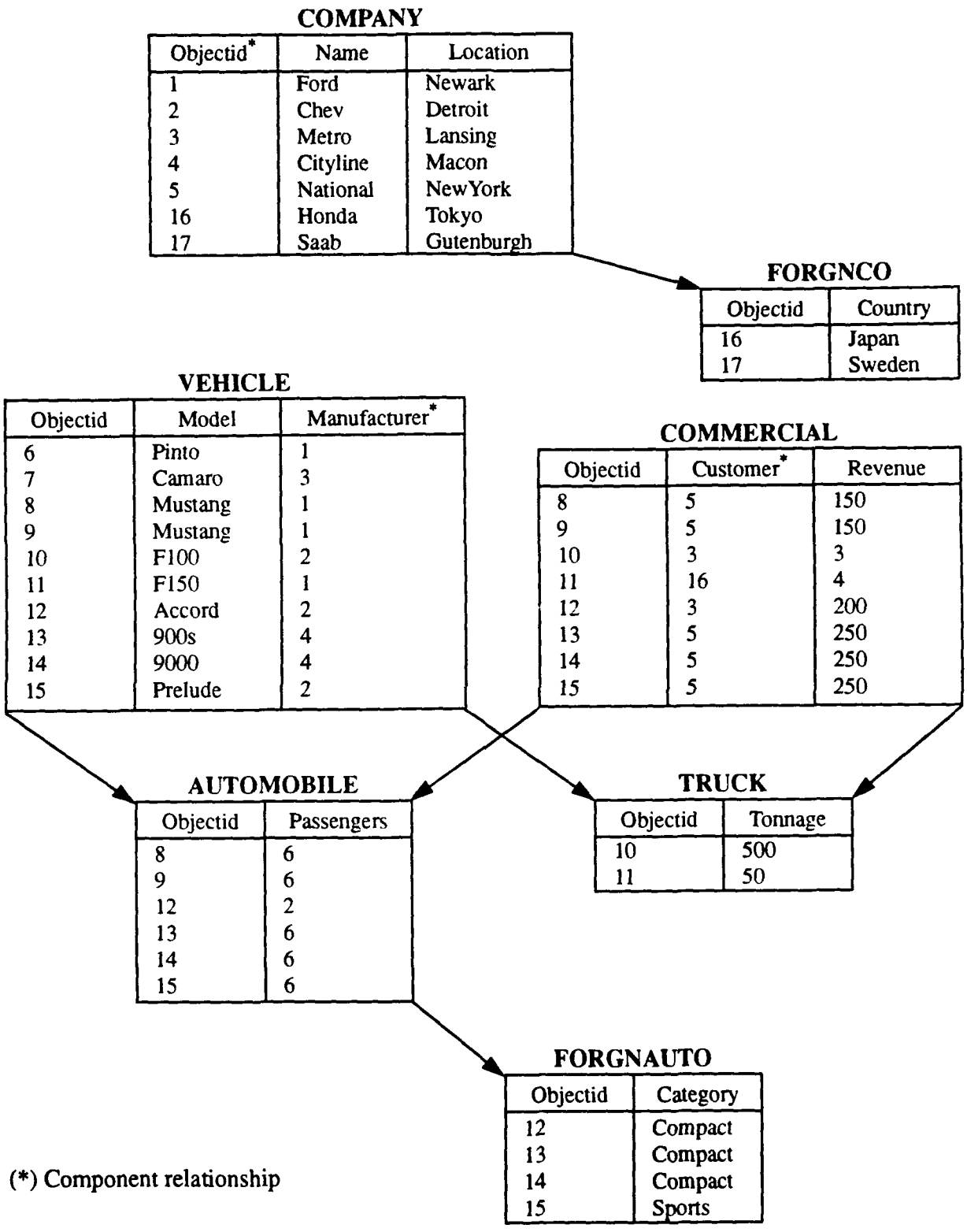


Figure 9. The Current Implementation Strategy Employed.

positive affect to the data management aspects are realized through more efficient class-instance retrievals. However, the class-instance retrievals require multiple class joins. We elaborate on the class-join issue in Chapter VI where the no-more-than-two-class-join limitation is discussed.

To handle the duplication issue of the second mapping strategy, we simply incur the increased storage expense. The storage expense associated with the duplication occurs because each object's *Objectid* is stored in their respective class instance. However, this does produce a minimum of storage overhead. The result of mapping the VEHICLE Class Hierarchy by using our mapping strategy is shown in Figure 9.

With the object-oriented-data-model-to-attribute-based-data-model mapping complete, the database system has a template for the data to be stored and accessed. The storing of data is accomplished, by the object-oriented data language INSERT statement. While the object-oriented data language RETRIEVE statement accesses the data. We continue the following section with mapping the object-oriented and attribute-based data languages.

2. Mapping the Object-Oriented Query

The are two types of object-oriented queries or transactions that we map to their attribute-based equivalents. The first is for the INSERT transaction and the second for the RETRIEVE query. Since data must be put into the database, we begin our discussion with the INSERT transaction.

The INSERT transaction is used to build the database. In object-oriented terms, the INSERT statement is used to build object instances, with each instance to be considered as a separate record. However, this record, containing all properties of an object, is not one all-encompassing record. It has several parts logically located in other class instances. However, the combining of an object's parts will produce the entire object instance [Elma89].

To begin inserting records (i.e., object data into its instance), the user must conform to the following object-oriented language syntax:

class_name.INSERT Objectid-value, attribute-value$_1$, attribute-value$_2$,..., attribute-value$_n$

The class_name is simply the name of the class for which an insert operation is requested. The dot notation, *class_name.INSERT*, indicates the command desired for that particular class. The

Objectid identifies each object uniquely and is distinct for each object instance. The remaining portion of the INSERT statement reflects the object instance's attribute values.

The multimodel/multilingual database system checks constantly the object-oriented schema when parsing the INSERT statement. This is necessary in order to place the object's attributes within the correct corresponding class. For instance, the Automobile subclass has two of its attributes, Model and Manufacturer, that are the common attributes among all other instances maintained in the Vehicle superclass. The multimodel/multilingual database system will check for the highest or top superclass within that object instance's class hierarchy. Once the top superclass has been identified, the components that apply to it are then stripped-off the object-oriented INSERT statement. These stripped-off attribute values are then mapped into their attribute-based data language-equivalent attributes.

After the attribute-based data language INSERT statement for the top superclass' attributes have been processed, the object-oriented interface code checks the object-oriented schema for other sublevel superclasses within the inserted objects class hierarchy. With each additional superclass, the appropriate attribute values are stripped off and mapped to an attribute-based data language INSERT-equivalent and processed. The object-oriented interface code continues to check for class-superclass relationships until the remaining object-oriented INSERT attributes can be mapped directly to the class for which the INSERT transaction takes place.

Again, once all superclass-instance-attribute requirements have been met, a final INSERT is generated for the specified class. The following example generates three attribute-based data language-equivalent statements and will help clarify the above object-oriented-to-attribute-based data-language INSERT mapping. For details of the mapping, please refer to Appendix B.

Automobile.INSERT 8, Mustang, 1, 5, 150, 6

[INSERT (<TEMP, Vehicle>,<Objectid, 8>,<Model, Mustang>,<Manufacturer, 1>)]

The first attribute-based data language INSERT statement for the subclass Automobile's superclass, Vehicle.

[INSERT (<TEMP, Commercial>,<Objectid, 8>,<Customer, 5>,<Revenue, 150>)]

The second attribute-based data language INSERT statement for the subclass Automobile's other superclass, Commercial.

[INSERT (<TEMP, Automobile>,<Objectid, 8>,<Passengers, 6>)]

The third attribute-based INSERT statement that fulfills the subclass Automobile's schema requirements.

In review, the above single object-oriented INSERT statement generated three separate attribute-based data language-equivalent INSERT statements. The user is unaware of the actual generation in the mapping. Once an object-oriented INSERT is entered using proper syntax, the multimodel/multilingual database system maps it to the attribute-based equivalent. The complete object-oriented database with its respective object-instance data will look similar to Figure 9. The one common element that is consistent throughout the class hierarchy and required to identify each distinct object instance is the Objectid. However, the user must maintain and keep unique each Objectid for all object instances. This is a break in standard object-oriented design with respect to an object's id from current commercial object-oriented database management systems (OODBMS). In the commercial systems, the Objectid is generated by the host database system. This alleviates the user from assigning and maintaining the large number of objectids. However, in our design, the user is responsible for each Objectid-to-object-instance assignment. This is done for practical purposes since our intent is not to duplicate all OODBMS features. Our intent is to demonstrate the feasibility of the object-oriented interface-approach to a common database. However, allowing the database system to generate all objectids for each object instance is more efficient and less prone to error. This system-generated objectid method is a future goal of the

multimodel/multilingual database system. With that in mind, the essence of our class hierarchy would not be realized without the proper objectid maintenance by the user.

Once the insertion of records into the database is complete, transactions against those data may begin. Our method of accessing the database is via the object-oriented RETRIEVE statement. All transactions against the database are performed using the following object-oriented RETRIEVE syntax.

class_name.RETRIEVE attribute-name(s) if condition

The dot notation for *class_name.RETRIEVE*, reflects the RETRIEVE operation for a particular class. While the *attribute-name* defines the attribute-values that are returned once the query condition has been satisfied. The *condition* includes conditions on the attributes of the class. There are three general cases used in mapping an object-oriented RETRIEVE to its attribute-based equivalent. Each case will be discussed with examples shown to better illustrate the mapping process. The following examples are derived from Appendix B.

General Case #1

Vehicle.RETRIEVE Model, Manufacturer, Objectid

The above RETRIEVE query requests that all *Model*, *Manufacturer*, and *Objectid* attribute values be returned for the class Vehicle. The attribute-based data language mapping of this RETRIEVE statement is as follows.

[RETRIEVE (TEMP=Vehicle) (Model, Manufacturer, Objectid)]

The class for which the RETRIEVE is made also happens to be the condition, *TEMP=Vehicle*. With the precondition consisting of the Vehicle class, and since no other

conditions are present, the attribute values for the *Model*, *Manufacturer*, and *Objectid* attributes are retrieved and returned to the user.

General Case #2

Vehicle.RETRIEVE Manufacturer.Name, Model

For this case, the class for which a retrieval is requested and the initial condition are the same, Vehicle. However, note that the dot notation is used again in the following expression, *Manufacturer.Name*. The translation of this dot notation is different from the command-type dot notation of General Case #2. This expression informs the system that another class is to be accessed, in this case, Company. The Company class has a component_of relationship with the Vehicle class through the Manufacturer attribute. The value requested from the class Company is the *Name* attribute-value associated with the attribute, *Manufacturer*, of class Vehicle. The multimodel/multilingual database system will check the schema to see if a component_of relationship exists. Since, in this case, the component_of relationship does exist, the Company *Name* attribute-value is returned as well as the vehicle's *Model* attribute-value. The equivalent attribute-based mapping is comprised of the following:

[RETRIEVE (TEMP=Vehicle) (Model)
COMMON (Manufacturer, Objectid)
RETRIEVE (TEMP=Company) (Name)].

The first RETRIEVE statement complies with the initial condition and also a portion of the query return value. The COMMON statement is similar to the join feature in the relational sense [Elma89] and is built into the kernel system for performing a two-class, e.g., in join on common attribute-values. The two common elements are the *Manufacturer* value of the first RETRIEVE and the *Objectid* value of the second RETRIEVE. The *Objectid* in the COMMON statement from the second RETRIEVE is obtained by the multimodel/multilingual database system and is not discussed in this thesis. However, these common features represent the pointers from one

class to another. The *Manufacturer* attribute-value points to the Company class while the Company class *Objectid* attribute-value points to the Manufacturer attribute in the class, Vehicle.

General Case #3

Vehicle.RETRIEVE Maunfacturer.Name, Model
if Manufacturer.Location = 'Boston'

This third general case is similar to general case #2 except for an additional condition. This additional conditional is the *Location* = *'Boston'*. The query requests all Company names and Vehicle model types for those vehicles with a Location attribute-value equal to *Boston*. The mapping translates to

[RETRIEVE (TEMP=Vehicle) (Model)
COMMON (Manufacturer, Objectid)
RETRIEVE ((TEMP=Company) and (Location='Boston')) (Name)]

This illustrates how a query with multiple conditions may be mapped to its attribute-based data language equivalent. There is no limit to the number of conditions as long as the classes and attributes exist for the appropriate class relationships.

Each of the above examples illustrates, in a general sense, the mapping of object-oriented-to-attribute-based queries. The user needs only be concerned with the syntax of the new object-oriented data language. We have deliberately kept this syntax simple and similar to the syntax of the attribute-based data language.

IV. BASIC IMPLEMENTAION ISSUES

The basic implementation for our object-oriented interface started with an implementation strategy. Once our strategy was formed we then created the data structures. These data structures were of two types: shared by all users and specific to each user. The third aspect of our implementation describes each multimodel/multilingual database system module. The four modules described are the language interface layer (LIL), the kernel mapping system (KMS), the kernel controller (KC), and the kernel formatting system (KFS).

A. OUR IMPLEMENTATION STRATEGY

Our implementation strategy consisted of three elements. The first element was to use a link-list data structure. Using linked-lists we are able to allocate memory dynamically.

The second element was to use similar system-module naming conventions. These naming conventions were the same for the other implemented interfaces, i.e., relational, hierarchical, and network. The third element was to use macro definitions within the source code. These definitions helped during the debugging process.

Each of the aforementioned implementation strategy elements enabled us to implement our object-oriented interface within three months. This quick implementation time was a result of following the previous data model/language interface's approach. Thereby allowing our interface implemented with complete system compatibility.

B. DATA STRUCTURES

The object-oriented data model/language interface was developed for a single user system. However, our interface may be updated to operate in a multi-user system. We propose two concepts for the data used in the language interface: (1) Data structures shared by all users, and (2) Data structures specific to each user. The reader must realize that the data structures used in our interface are generic to the multimodel/multilingual database

system. Hence, these same structures support our interface, as well as the relational, hierarchical and network. The following data structures are provided in a schematic format in Appendix C.

1. Data Structures Shared by all Users

The data structures shared by all users of the multimodel/multilingual database system originate from the object-oriented database schemas. They consist of classes, superclasses, subclasses and attributes. These are not only shared by all users, but also are shared by the four modules of the interface, i.e. LIL, KMS, KC, KFS. Figure 10 depicts the first data structure used to maintain data. This structure represents a union. Hence, it is generic because a user can utilize this structure to support our object-oriented interface as well as the other interfaces. However, we concentrate on the object-oriented data model/ language interface..

```
union dbid_node {
    struct rel_dbid_node *dn_rel;
    struct hie_dbid_node *dn_hie;
    struct net_dbid_node *dn_net;
    struct ent_dbid_node *dn_fun;
    struct obj_dbid_node *dn_obj; }
```

Figure 10. The dbid_node Data Structure

The last field of the *dibi_node* data structure points to a record that contains information about an object-oriented database.

Figure 11 illustrates the struct definition for the record mentioned above. The first field is a character array containing the name of the object-oriented database. The next field contains an integer value representing the number of classes in the database. The third and

fourth fields are pointers to other records containing information about each class in the database. The final field is a pointer to the next object-oriented database schema.

```
struct obj_dbid_node {
    char                odn_name[DBNLength + 1];
    int                 odn_num_cls;
    struct ocls_node    *odn_first_cls;
    struct ocls_node    *odn_curr_cls;
    struct obj_dbid_node *odn_next_db; }
```

Figure 11. The obj_dbid_node Data Structure

The record *ocls_node* contains information about each class in the database. See Figure 12. This structure is organized similar to the *obj_dbid_node* structure. The first field of the record holds the name of the class. The next three fields contain the number of attributes, the number of super classes and the number of subclasses of this class respectively.

```
struct ocls_node {
    char                ocn_name[RNLength + 1];
    int                 ocn_num_attr;
    int                 ocn_supcls;
    int                 ocn_subcls;
    struct o_supcls_node *ocn_first_supcls;
    struct o_supcls_node *ocn_curr_supcls;
    struct o_subcls_node *ocn_first_subcls;
    struct o_subcls_node *ocn_curr_subcls;
    struct oattr_node   *ocn_first_attr;
    struct oattr_node   *ocn_curr_attr;
    struct ocls_node    *ocn_next_cls; }
```

Figure 12. The ocls_node Data Structure

The next two pointers point to the superclass records. Each superclass record holds the name of the super class. a pointer to that class and a pointer to the next super class.

See Figure 13. The seventh and eight fields are pointers to the subclass records. See Figure 14. The next two are pointers to the records for the class attributes. See Figure 15. The last field points to the next class in the database.

```
struct o_supcls_node {
    char                  osn_name[RNLength + 1];
    struct ocls_node      *osn_supcls;
    struct o_supcls_node  *osn_next_supcls; }
```

Figure 13. The o_supcls_node Data Structure

The *o_supcls_node* data structure allows us to handle multiple inheritance by forming a list of superclasses. The same argument applies to the *o_subcls_node* data structure, but this allows for more than one subclass of a class.

```
struct o_subcls_node {
    char                  osn_name[RNLength + 1];
    struct ocls_node      *osn_subcls;
    struct o_subcls_node  *osn_next_subcls; }
```

Figure 14. The o_subcls_node Data Structure

Figure 16 shows the final structure used to support the definition of the object-oriented database schema. The first field is an array which holds the name of the attribute. The second field determines the attribute type. An attribute type can either be a class name (representing a composite attribute), integer, float or character.

```
struct oattr_node {
    char                  oan_name[ANLength + 1];
    char                  oan_type[RNLength + 1];
    int                   oan_length;
    struct oattr_node     *oan_next_attr; }
```

Figure 15. The oattr_node Data Structure

The next field is the character attribute-value maximum length. The last field points to the next attribute record of this class.

2. Data Structures Specific to each User

This category of data represents information needed to support each user's particular interface needs. The data structures used to accomplish this form a hierarchy. The root of this hierarchy is defined by the structure *user_info*. This structure maintains information on all of the current users of a particular language interface. See Figure 16. The *user_info* structure holds the user id, a union that describes a particular interface and a pointer to the next user.

```
struct user_info {
    char                ui_uid[UIDLength + 1];
    union li_info       ui_li_type;
    struct user_info    *ui_next_user; }
```

Figure 16. The user_info Data Structure

The union that describes a particular interface is depicted in Figure 17. In Figure 17, our concern is for the data structures which contain information pertaining to each object-oriented-interface user..

```
union li_info {
    struct sql_info     li_sql;
    struct dli_info     li_dli;
    struct dml_info     li_dml;
    struct dap_info     li_dap;
    struct ool_info     li_ool;
```

Figure 17. The li_info Data Structure

The first field of the *ool_info* structure is a record. It contains the current information on the database being accessed. The second field contains the file descriptor

and file identifier of a file for object-oriented interface transactions, i.e. either queries or schema definitions.

```
struct ool_info {
    struct curr_db_info   oi_curr_db;
    struct file_info      oi_file;
    struct tran_info      oi_ool_tran;
    struct ddl_info       *oi_ddl_files;
    struct tran_info      *oi_abdl_tran;
    union  kms_info       oi_kms_data;
    union  kfs_info       oi_kfs_data;
    union  kc_info        oi_kc_data;
    int                   oi_answer;
    int                   oi_error;
    int                   oi_operation; }
```

Figure 18. The ool_info Data Structure

The next field is a structure which holds information describing the transactions to be processed. The information includes the number of requests, the first request and the current request to be processed. The fourth field is a pointer to a structure describing the descriptor file and template file. These files contain information about the attribute-based data language schema corresponding to the current object-oriented database schema. The pointer *oi_abdl_tran* points to a record that describes the equivalent attribute-based data language transactions written in our object-oriented data language, i.e., the translated object-oriented data language requests. The data for the pointer is provided by KMS and used by KC. The next three fields are unions that contain information required by KMS, KFS and KC respectively. The *oi_answer* holds the menu choice the user has chosen. The *oi_error* holds any error type which occurred during query processing. The *oi_operation* indicates the operation to be performed, i.e., loading a new object-oriented database or executing a request against an existing object-oriented database. In the latter case, it indicates which attribute-based data language request to be executed.

C. THE DESCRIPTION OF THE MODEL ALGORITHMS

The algorithms used in each module, LIL, KMS, KC and KFS, are described next. Also, we describe what each function does.

1. Language Interface Layer (LIL)

The LIL module is used to control the order in which the other modules are called. It allows the user to enter transactions by either a file input or by terminal entry. A transaction either defines a database schema or is a query against an existing object-oriented database. The mapping process begin when LIL sends a single transaction to KMS. After KMS parses and constructs an equivalent attribute-based-data-language-interface it is sent to KC for processing. Control always returns to LIL. At this point, the user may close the session by exiting to the operating system. When the system is first run, the user is provided with the multimodel/multilingual database system main menu. From the menu, the user chooses a particular data model/language interface. For our purposes we assume the object-oriented data model/language interface is chosen. Once the object-oriented interface is chosen, LIL is activated and calls the function *ool_main*.

ool_main(): A new *user_info* structure is allocated and initialized by calling the function *new_obj_user()*. Control is then transferred to *o_language_interface_layer()*.

o_language_interface_layer(): This function displays a menu for either defining a new database, processing an existing object-oriented database or exiting the object-oriented interface. Depending on the user's choice, it calls either *o_load_new()*, *o_process_old()* or *o_save_catalogs()*.

o_load_new(): The name of the new database is requested and checked to see if it already exists in the list of existing object-oriented databases. If it does not exist, it appends a new *obj_dbid_node* to the list and initializes. The user-input mode for the database schema definition is determined as either file input or terminal entry. The database definitions are read and stored in the *obj_req_info* structure and parsed by KMS. It then

calls *o_creates_to_KMS()*, *o_build_ddl_files()* and *o_Kernel_Controller()* sequentially as long as no error exists in the process.

o_process_old(): This function asks for the name of the database and checks if it exists either in the list of schemas or as a catalog file. If found, the current pointers are set to this database. The *o_process_old()* function displays a menu for either mass loading data from a file, entering queries from a file, entering queries from the terminal or displaying the current database schema. If the user wants to issue requests, they are read and stored in *obj_req_info* structure to be parsed by KMS. Depending on the user's choice, it calls either *o_mass_load()*, *o_queries_to_KMS()* or *o_display_schema()*.

o_save_catalogs(): This function executes upon exiting the object-oriented interface session. Thereby the database schemas in the list are saved into separate catalog files, the.*databaseName.cat*. They are also saved in a fixed format to be used in later sessions. Finally, the associated memory from the previous object-oriented interface session is de-allocated.

o_build_ddl_files(): It creates two multimodel/multilingual database system files, the template file, *databaseName.t*, and the descriptor file, *databaseName.d*. The template file is the attribute based schema corresponding to the object-oriented schema. While the descriptor file contains information on the attributes that are used for data clustering.

o_creates_to_KMS(): This routine sends the list of database definitions stored in the *obj_req_info* structures. Each is sent one by one to KMS by calling *ool_kernel_mapping_system()*. Whereby the object-oriented schema is stored into the data structures described in Figures 11 through 15.

o_queries_to_KMS(): This routine lists the queries onto the screen. The selection menu is then displayed allowing one of the queries to be selected. The selected query is then sent to KMS for parsing. Once parsed, it creates the corresponding attribute-based data language query. The *o_Kernel_Controller()* function is then called to execute the corresponding attribute-based data language query.

o_mass_load(): It reads a user-generated data file and creates the attribute-based data language INSERT statements. It writes those INSERT statements into the file.*TransFile*. It also ensures each are correct before sending them to the kernel system. If no error occurs, each INSERT statement is sent to the kernel system by calling *TI_S$TrafUnit()* to be stored on the disk. After each insert is sent to the kernel system, *ool_chk_responses_left()* is called to get the response from the kernel system before the next one is issued. Upon completion, *.TransFile* is deleted.

2. Kernel Mapping System (KMS)

KMS has two functions: (1) parse the object-oriented data language request to validate the syntax, and (2) translate, or map, the request to an equivalent attribute-based data language request. Once an equivalent attribute-based data language request is formed, it is made available to KC. KC then processes the request for kernel-system execution.

ool_kernel_mapping_system(): If the request is a database schema definition, it calls *construct_schema()*. Otherwise, for a data manipulation request, the routine *translate_request()* is called.

construct_schema(): This function parses the object-oriented data language database descriptions and creates the object-oriented schema. Once created it is stored in the data structures described in Figures 11 through 15.

translate_request(): It first calls *parse_ool_request()*. If there is no error in the request, then *construct_abdl_request()* is called to map the object-oriented data language request to attribute-based data language equivalent.

parse_ool_request(): It parses the object-oriented data language request and checks for correct syntax. Also, it stores the necessary mapping-process information into the *obj_kms_info* structure. However, the INSERT requests are excluded. For INSERT requests, the data in the request is parsed and written in mass-load-file format and placed in the file *.insert_file*.

construct_abdl_request(): It generates attribute-based data language requests which correspond to object-oriented data language requests. It uses the information stored in the *obj_kms_info* structure, except the INSERT requests. However, *o_mass_load()* function is called for the INSERT request, then the file *.insert_file* is deleted.

3. Kernel Controller (KC)

KC submits the attribute-based data language transaction(s) to the kernel system for processing. If the transaction involves a database schema definition, an insertion, a delete request or an update, then control is returned to LIL after the kernel system processes the transaction. If it involves a retrieve request, KC sends the translated attribute-based data language request to the kernel system, receives the results back, loads the results into a buffer and calls KFS to format/display the results. Control returns to LIL.

o_Kernel_Controller(): This function routes control to the relevant functions by checking the *oi_operation* field of the *ool_info* data structure. For a database definition operation, *o_load_tables()* is called. For a database manipulation operation, *ool_req_execute()* is called.

o_load_tables(): This routine loads the database template file. This file is the attribute-based schema which corresponds to the object-oriented schema. The template file is opened and the test interface (TI) function *dbl_template()* is called. This TI function reads the template file and loads it onto the disk as the meta data. The file is then closed and control is returned to the *o_Kernel_Controller()* routine. This routine returns control back to LIL.

ool_req_execute(): It sends the translated attribute-based data language request to the kernel system using *TI_S$TrafUnit()* defined in TI. It then calls *ool_chk_res_left()* to ensure all requests have been processed and the results from the kernel system have been received.

ool_chk_responses_left(): This function communicates with the kernel system and receives the message about the condition of the request. If there are errors, the

TI_R$ErrorMessage() is called to get the error message. The function *TI_ErrRes_output()* is called to display the error message. However, if no error occurred, *TI_R$ReqRes()* is called to receive the response from the kernel system. The response buffer is then checked to see if this is the last response. If it is the last response of a retrieve request, the results are sent to KFS by calling *o_kernel_formatting_system()*. For either an insert, delete or update request, control is transferred back to LIL.

4. Kernel Formatting System (KFS)

KFS is called from KC when KC obtains the final results of a retrieve request from the kernel system. The results are passed to KFS in one or more character buffers, called response buffers. KFS manipulates the contents of these buffer(s) and displays the results in a table format. KFS uses the *kfs_obj_info* data structure for temporary storage.

o_kernel_formatting_system(): This function processes the response buffer and displays the results of a retrieve request in a table format. The response buffer from the kernel system is a long character string where each token is separated by a null byte character. Each returned-buffer value is proceeded by its attribute name. This function also parses the string and displays the first set of attribute names as table-cell headings. It then stores their values in the *kfs_obj_info* data structure. Finally, it outputs those stored values and continues to parse displaying the next values while ignoring the other attribute names.

V. OTHER IMPLEMENTATION ISSUES

The implementation of our interface for the multimodel/multilingual database system has been described. We now focus on two other implementation concerns. First, we describe how to change the object-oriented schema. Second, the user-defined, class-definition, actions are discussed. Each of these two implementation issues will help the user maintain a current model for their modeled application.

A. SCHEMA MODIFICATIONS

Changes to our object-oriented schema may be of two types; (1) Adding, deleting or renaming classes and/or their properties, or (2) Modifying the class relationships within the class hierarchy. Both schema-change types are reconfigure the schema file to better represent the new application model. This process of schema-file modification is not difficult but requires the user to be cognizant of all class and class hierarchical interactions. We discuss both schema change methods together since each are accomplished by a similar process.

The schema modification process, for both change types, requires the user to modify the schema file. The schema file contains the logical description of the object-oriented database structure. The database structure is changed by editing the appropriate components of the class definition. These components fall into two categories. One category describes the class behavior and it fulfills the first schema-modification type. Since the behavior of a class is determined by its properties, changes to class properties may affect subclasses within the class hierarchy. For instance, changing an attribute in a superclass affects the subclass because of the inheritance principle. All class-property changes require careful consideration in their execution.

The second category of class-definition changes fulfills the second type of schema modification, modifying the class relationships. A class' relationship is identified in the class definition by the components, SUPCLASS and SUBCLASS. These components

identify a class' relationship relative to the other classes within the class hierarchy. The user modifies the class hierarchy by either adding, deleting or renaming a class. Adding a class to the class hierarchy requires a new class name, its relationship to other classes, and its class properties. The important aspect of adding a new class requires identifying its proper class hierarchical relationship relative to the other classes.

Deleting a class has several implications. The first implication determines if the class to delete is a leaf class (a leaf class is one that has no subclasses). If it is a leaf class, then deleting it does not affect any other class. The user may delete the class without it affecting any other class within the class hierarchy. However, if it is not a leaf class and has one or more subclasses, then the deletion becomes more complicated. This complication results from those class properties that the subclasses of the soon-to-be-deleted class inherits. The user must identify what properties are inherited by the subclass or subclasses. Some or all inherited class properties may be required to maintain the modeled application's structure regarding the one or more subclasses. If the class properties are required, then the issue to keep those class properties needs to be addressed. Since the inherited class properties are necessary, the user must reconfigure either the superclass' or the subclass' class definition. If the class properties are not required, the SUPCLASS and SUBCLASS attribute-values are modified to reflect the change.

Renaming a class within the class hierarchy requires the class-definition class name to be changed. Since the class name changes, the user must propagate this name change throughout its affected hierarchy. No class property changes are necessary.

To summarize, the class behavior and class relationship schema modifications are realized by editing the object-oriented schema file. Once this file's modification is complete, the object-oriented-schema-to-attribute-based-schema mapping must occur. Since the schema was changed, the data, or instances, must also be changed. These instance changes are done by reinserting all data, in the new schema format, into the database.

B. USER-DEFINED OPERATIONS

To fully realize the object-oriented design, user-defined class operations should be incorporated into the class definitions. User-defined operations are specific to a given class. The class uses these operations to interact with the other classes within the class hierarchy. However, our implementation does not include these action in the class definition. We assume the two implemented object-oriented data language actions, INSERT and RETRIEVE, are part of each class definition. Our intent is to see these actions included in the class definition. This would allow each class a true external interface capability. Since this class external interface is part of the class definition, the encapsulation principle is maintained. Through this principle, the class modular design is realized. Thus, the object-oriented design features and constructs are complete.

However, before user-defined actions become part of the class definition a more comprehensive object-oriented data language token-parsing routine is needed. This parsing routine must be modified to handle the various object-oriented data language syntax additions. The syntax additions would be those specific to each new user-defined class action.

C. OBJECTID MAINTENANCE

For our implementation, the user generates and maintains each object's objectid. This method of objectid maintenance is sufficient to maintain our small test database. Also, our Vehicle-class application only has seven classes each with one to three attributes. This number of classes combined with the 16 class instances does not overwhelm the user. However, for larger databases with multiple object-oriented application, the objectid-maintenance task is more difficult. This difficulty stems from maintaining all the class instances as well as their class-hierarchical relationships. To generate and maintain each object's unique identity, or objectid, it should be done by the database system.

For future object-oriented interface updates, we suggest the multimodel/multilingual database system generate and maintain all objectids. This would alleviate the user from

becoming too entrenched with the data maintenance. The user could then concentrate on the data management.

This change, to allow system-objectid maintenance, could be made by updating the object-oriented-database-records file. The update would consist of prompting the user for each object instance and its associated properties. Once entered, the system would assign a sequential objectid to each object instance. This system-generated objectid would then be stored with the object itself.

VI. OUR CONCLUSION

Our object-oriented interface for the multimodel/multilingual database system is a viable and effective alternative to database design. Through this alternative design, we successfully implement the important object-oriented data model's features and constructs. These features and constructs include, but are not limited to, generalization/specialization, inheritance, class encapsulation, and object reusability. When combined, these features and constructs uniquely identify the object-oriented data model.

We incorporate these object-oriented features and constructs into a multi-*modeled* environment sharing a common database. This common-database sharing is a working example of a single-system environment. Since this single-system environment exists, our resources are consolidated and there is no data duplication. Thus, our object-oriented interface not only fulfills the object-oriented data model requirements, but also meets those aspects of single-system operations.

Our interface, while fulfilling the object-oriented and single-system requirements, also was developed with minimal costs. The developmental costs associated with building the interface included the time to develop an object-oriented schema and the writing of the interface source code. To obtain the schema, we modeled a Vehicle-hierarchy application. This application was then formed into a generalized and specialized class hierarchy. From the Vehicle class hierarchy, or the conceptual view, we converted it into our general object-oriented schema. Once this object-oriented schema was completed, we mapped it to the attribute-based schema, or the logical view.

Writing the interface source code was accomplished in three months. Compare these three months to develop our interface with the time needed to design, develop and implement a stand-alone object-oriented database system, and our approach is much more appealling.

However, there were three limitations to our design. First, there is no standard object-oriented data model/language. Second, there was a two-class join limitation. Finally, our object-oriented interface was not programmed in an object-oriented programming language. Each of these three limitations is discussed below.

We conclude our thesis with prospects for future research.

A. LIMITATIONS

The three limitations to our interface, lack of a standard object-oriented data model/ language, the two-class join limit, and using a non-object-oriented programming language, did not hinder our implementation. The lack of a standard object-oriented data model/ language allowed us to create our own data model/language. Our object-oriented data model/language incorporated the necessary object-oriented principles. These principles were borrowed from other existing data models/languages, i.e., relational and hierarchical. Incorporating these borrowed principles into one data model/language resulted in our object-oriented data model/language.

The second limitation, maximum two-class join, was due to the multimodel/ multilingual database system. However, our goal was not to produce a production system. Our goal was to build a demonstrable object-oriented interface for the multimodel/ multilingual database system. We successfully accomplished this interface. Hence, our goal was obtained.

The third limitation, not programming in an object-oriented language, did force us to change our implementation strategy. Our initial strategy was to use the inherent object-oriented features of an object-oriented programming language, i.e., class encapsulation, inheritance, and user-defined class operations [Lipp92]. However, the multimodel/ multilingual database system could not compile C++ (our intended programming language) source code. Thus, we used the system compatible programming language, C. Using the system compatible language C proved a difficult task to implement our object-oriented features and constructs. However, each feature and construct was incorporated by

manipulating the C programming language features. Along with manipulating C's features we relied on complicated link-list data structures.

B. FUTURE RESEARCH

There are two issues for future research. The first is to incorporate the covering, or aggregation, principle. The second would be to code our object-oriented interface in an object-oriented programming language, e.g., C++.

The covering, or aggregation, principle links two separate and distinct class hierarchies. Through this link, a class from one hierarchy could access a class from the other class hierarchy. The problem is to create this link between the two class hierarchies. The key to creating this link is to identify a common element between the two class hierarchies. Once this link is identifed, the covering principle may be realized.

The second research issue would require the multimodel/multilingual database system ported over to an object-oriented-programming-language-compatible system. Once the multimodel/multilingual database system has been ported over to a compatible system , the interface could be rewritten in an object-oriented programming language. Hence, our object-oriented data model/language features and constructs would be fully supported.

However, the problem associated with converting from a non-object-oriented programming language to an object-oriented one, is configuring the new system whereby the multimodel/multilingual database system would reside. Once resolved, and the covering principle incorporated, the object-oriented interface for the multimodel/ multilingual database system would be complete.

APPENDIX A

OBJECT-ORIENTED FEATURES AND CONSTRUCTS

Object: The user's view of a real world entity.

Objectid: The user generated unique identifier for each object.

Class: A set of objects. The basis of the schema hierarchy.

Object-Oriented Database Schema (schema): The database descriptor for its objects and is described by the set of class definitions connected by the super-class/subclass hierarchical relationships and includes the class description.

Class Definition: The features that make up a class and include the following:

Class_Name
Object_ID
Class_Relationship
Attributes
(Actions, inherited from Base or Root class and are standard throughout system)

Class_Name: The name assigned to a particular class.

Class_Relationship: The relationships between the specified class and other existing classes. May include a-kind-of and component_of relationship representations.

a-kind-of -- a similar representation of the object specified but with unique attributes and actions of its own.

component_of -- the relationship between an attribute of one class and that of the newly defined attribute class.

Attributes/Instance Variables: The variables that make up the specific elements of a class. A class may have an attribute which is itself a separate class.

Actions: Operations that may be performed on a given class, object, or set of objects. Mainly concerned with object manipulation.
class Action or object Action.

Class_Action - operations that are applied to a class and may

be unique for each class as well as inherited
from classes higher in the hierarchy.
Examples include:
class_name.RETRIEVE - retrieve a particular class
instance.
class_name.INSERT - insert a new class by providing
information on the following:
Class_Name, Class_Relationship,
Class_Attributes, Class_Actions.
class_name.UPDATE - after retrieving a class, make
changes to its name, attributes,
and actions. Further checks will
need to be incorporated to prevent
inadvertent modification of
attributes/actions that are
inherited by subclasses.
class_name.DELETE - the deletion of a class by one
of two options:
1. delete all instance and subclasses
2. delete class only and
"reconnect" dangling subclasses/
instances.
Similar checks to those mentioned
in class.UPDATE will need to be made.

Class Hierarchy: The organization of classes into a hierarchy and the aggregation of classes to form its composition.

Inheritance: Objects in a class inherit properties (Attributes, Actions) from classes higher in the class hierarchy.

Encapsulation: Each object has its own holding state (in affect, memory) through incorporation of object attributes and actions.

Covering: One class is a cover of another class if every object of the first class corresponds to a subset of objects of the second class.

Object-Oriented Data Model: The conceptual representation of the Object-Oriented features listed above and includes how the user may view his/her particular application/world schema.

Object-Oriented Data Language: The language through which the Object-Oriented Data Model is accessed in order to retrieve data from the database.

Attribute Based Data Language: The kernel language inherent to the Multibackend Database Supercomputer.

APPENDIX B

THE OBJECT-ORIENTED INTERFACE USER'S MANUAL

A. OVERVIEW

The object-oriented data model/language interface allows the user to input transactions from either a file or the terminal. A transaction can take the form of either database schema definitions or queries against an existing object-oriented database. The object-oriented data model/language interface is menu-driven. Each menu prompts the user for additional transaction information. If the transactions are database schema definitions, they are processed automatically by the system. If the transactions are queries, the user will be prompted by another menu to selectively pick an individual query to be processed. The user also has the option to return to a previous menu within the menu hierarchy.

B. USING THE SYSTEM

The user may perform two operations: they can either define a new database schema or they can manipulate an existing database. The first menu, shown below, prompts the user for the function to perform. This menu, MENU1, looks like the following:

Enter type of operation desired
(l) - load a new database
(p) - process existing database
(x) - return to the MLDS/MDBS system menu

ACTION ---->

Upon selecting the desired operation, the user is prompted to enter the name of the database. For the load operation, the database name can not be one presently in use. Likewise, for a process-existing operation, the database name provided must exist in the database. The session continues once a valid database name has been entered.

The load operation was selected from MENU1, the second menu, MENU2a, requests the mode of transaction input. The input may come from a file or be interactive from the

terminal. The file option is used for long transactions. This helps to avoid any errors in the transaction. The MENU2a looks like the following:

Enter mode of input desired
- **(f) - read in a group of creates from a file**
- **(t) - read in creates from the terminal**
- **(x) - return to the main menu**

ACTION ---->

For the MENU1 process-existing-database operation, MENU2b is displayed next and looks like the following:

Enter your choice
- **(d) - display schema**
- **(m) - mass load from a data file**
- **(f) - read in a group of queries from a file**
- **(t) - read in queries from the terminal**
- **(x) - return to previous menu**

ACTION ---->

In either MENU2a or MENU2b, the user is prompted for the name of the file, but only if the file option is selected. If the terminal option is selected, a message is displayed to remind the user of the correct transaction-entry format. If the user wants to see the current object-oriented schema, they do so from MENU2b. This menu also allows the data to be loaded from a data file. We continue with describing the database definitions and manipulations processes.

1. Processing Database Definitions

When the user has specified the file name of database definitions or entered them from the terminal, the system processes these definitions and creates the template file. If there is a descriptor file for this database already, the user is asked to either use it or to create another one. If the user wants to create another one, they may define the clustering

attributes or allow the system to define its own clustering information. Control is returned to MENU1 to allow the user pick a new operation.

2. Processing Queries

When the user has specified the file name of queries or has entered them from the terminal, the queries are displayed to the screen. When queries are listed to the screen from the transaction list, a number is assigned, starting with number one, to each query in ascending order. The number is displayed beside the first line of each query. Next, an access menu, MENU3, is displayed which looks like the following:

Pick the number or letter of the action desired
(num) - execute one of the preceeding queries
(d) - redisplay the file of queries
(x) - return to the previous menu

ACTION ---->

Since the displayed queries might exceed the vertical height of the screen, only a screen full of queries are displayed at a time. The next page of queries can be viewed by hitting the space key. The order in which the queries are listed is not significant. However, they may be selected in any order for execution. Unlike processing the database definitions, control returns to MENU2b. This is because the user may have more than one file of queries to process against a particular database. They may also wish to input some extra queries directly from the terminal. Once the user has finished processing a particular database, they can exit back to MENU1 to either change operations or exit to the operating system.

Some sample INSERT statements and their corresponding attribute-based data language equivalents that create the VEHICLES database is provided in section C. The sample RETRIEVE statements against the VEHICLE database, their corresponding attribute-based data language equivalents and their results are provided in section D.

C. SAMPLE INSERT STATEMENTS

1) company.insert 1, Ford, Newark
[INSERT (<TEMP, Company>, <OBJECTID, 1>, <NAME, Ford>, <LOCATION, Newark>)]

2) company.insert 2, Chev, Detroit
[INSERT (<TEMP, Company>, <OBJECTID, 2>, <NAME, Chev>, <LOCATION, Detroit>)]

3) company.insert 3, Metro, Lansing
[INSERT (<TEMP, Company>, <OBJECTID, 3>, <NAME, Metro>, <LOCATION, Lansing>)]

4) company.insert 4, Cityline, Macon
[INSERT (<TEMP, Company>, <OBJECTID, 4>, <NAME, Cityline>, <LOCATION, Macon>)]

5) company.insert 5, National, NewYork
[INSERT (<TEMP, Company>, <OBJECTID, 5>, <NAME, National>, <LOCATION, Newyork>)]

6) vehicle.insert 6, Pinto, 1
[INSERT (<TEMP, Vehicle>, <OBJECTID, 6>, <MODEL, Pinto>, <MANUFACTURER, 1>)]

7) vehicle.insert 7, Camaro, 3
[INSERT (<TEMP, Vehicle>, <OBJECTID, 7>, <MODEL, Camaro>, <MANUFACTURER, 3>)]

8) automobile.insert 8, Mustang, 1, 5, 150, 6
[INSERT (<TEMP, Vehicle>, <OBJECTID, 8>, <MODEL, Mustang>, <MANUFACTURER, 1>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 8>, <CUSTOMER, 5>, <REVENUE, 150>)]
[INSERT (<TEMP, Automobile>, <OBJECTID, 8>, <PASSENGERS, 6>)]

9) automobile.insert 9, Mustang, 1, 5, 150, 6
[INSERT (<TEMP, Vehicle>, <OBJECTID, 9>, <MODEL, Mustang>, <MANUFACTURER, 1>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 9>, <CUSTOMER, 5>, <REVENUE, 150>)]
[INSERT (<TEMP, Automobile>, <OBJECTID, 9>, <PASSENGERS, 6>)]

10) truck.insert 10, F100, 2, 3, 3, 500
[INSERT (<TEMP, Vehicle>, <OBJECTID, 10>, <MODEL, F100>, <MANUFACTURER, 2>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 10>, <CUSTOMER, 3>, <REVENUE, 3>)]
[INSERT (<TEMP, Truck>, <OBJECTID, 10>, <TONNAGE, 500>)]

11) truck.insert 11, F150, 1, 16, 4, 50
[INSERT (<TEMP, Vehicle>, <OBJECTID, 11>, <MODEL, F150>, <MANUFACTURER, 1>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 11>, <CUSTOMER, 16>, <REVENUE, 4>)]
[INSERT (<TEMP, Truck>, <OBJECTID, 11>, <TONNAGE, 50>)]
12) forgnauto.insert 12, Accord, 2, 3, 200, 2, Compact

[INSERT (<TEMP, Vehicle>, <OBJECTID, 12>, <MODEL, Accord>, <MANUFACTURER, 2>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 12>, <CUSTOMER, 3>, <REVENUE, 200>)]
[INSERT (<TEMP, Automobile>, <OBJECTID, 12>, <PASSENGERS, 2>)]
[INSERT (<TEMP, Forgnauto>, <OBJECTID, 12>, <CATEGORY, Compact>)]

13) forgnauto.insert 13, 900s, 4, 5, 250, 6, Compact
[INSERT (<TEMP, Vehicle>, <OBJECTID, 13>, <MODEL, 900s>, <MANUFACTURER, 4>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 13>, <CUSTOMER, 5>, <REVENUE, 250>)]
[INSERT (<TEMP, Automobile>, <OBJECTID, 13>, <PASSENGERS, 6>)]
[INSERT (<TEMP, Forgnauto>, <OBJECTID, 13>, <CATEGORY, Compact>)]

14) forgnauto.insert 14, 9000, 4, 5, 250, 6, Compact
[INSERT (<TEMP, Vehicle>, <OBJECTID, 14>, <MODEL, 9000>, <MANUFACTURER, 4>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 14>, <CUSTOMER, 5>, <REVENUE, 250>)]
[INSERT (<TEMP, Automobile>, <OBJECTID, 14>, <PASSENGERS, 6>)]
[INSERT (<TEMP, Forgnauto>, <OBJECTID, 14>, <CATEGORY, Compact>)]

15) forgnauto.insert 15, Prelude, 2, 5, 250, 6, Sports
[INSERT (<TEMP, Vehicle>, <OBJECTID, 15>, <MODEL, Prelude>, <MANUFACTURER, 2>)]
[INSERT (<TEMP, Commercial>, <OBJECTID, 15>, <CUSTOMER, 5>, <REVENUE, 250>)]
[INSERT (<TEMP, Automobile>, <OBJECTID, 15>, <PASSENGERS, 6>)]
[INSERT (<TEMP, Forgnauto>, <OBJECTID, 15>, <CATEGORY, Sports>)]

16) forgnco.insert 16, Honda, Tokyo, Japan
[INSERT (<TEMP, Company>, <OBJECTID, 16>, <NAME, Honda>, <LOCATION, Tokyo>)]
[INSERT (<TEMP, Forgnco>, <OBJECTID, 16>, <COUNTRY, Japan>)]

17) forgnco.insert 17, Saab, Gutenburgh, Sweden
[INSERT (<TEMP, Company>, <OBJECTID, 17>, <NAME, Saab>, <LOCATION, Gutenburgh>)]
[INSERT (<TEMP, Forgnco>, <OBJECTID, 17>, <COUNTRY, Sweden>)]

D. SAMPLE RETRIEVE STATEMENTS

1) vehicle.retrieve manufacturer, model, objectid

[RETRIEVE (TEMP=Vehicle) (OBJECTID, MODEL, MANUFACTURER)]

OBJECTID	MODEL	MANUFACTURER
6	Pinto	1
7	Camaro	3
8	Mustang	1
9	Mustang	1
10	F100	2
11	F150	1
12	Accord	2
13	900s	4
14	9000	4
15	Prelude	2

2) truck.retrieve manufacturer, model, objectid, tonnage

[RETRIEVE (TEMP=Truck) (TONNAGE, OBJECTID)
COMMON (OBJECTID, OBJECTID)
RETRIEVE (TEMP=Vehicle) (MODEL, MANUFACTURER)]

TONNAGE	OBJECTID	MODEL	MANUFACTURER
500	10	F100	2
50	11	F150	1

3) truck.retrieve revenue, customer, objectid, tonnage

[RETRIEVE (TEMP=Truck) (TONNAGE, OBJECTID)
COMMON (OBJECTID, OBJECTID)
RETRIEVE (TEMP=Commercial) (CUSTOMER, REVENUE)]

TONNAGE	OBJECTID	CUSTOMER	REVENUE
500	10	3	3
50	11	16	4

4) automobile.retrieve manufacturer, model, objectid, passengers

[RETRIEVE (TEMP=Automobile) (PASSENGERS, OBJECTID)
COMMON (OBJECTID, OBJECTID)
RETRIEVE (TEMP=Vehicle) (MODEL, MANUFACTURER)]

PASSENGERS	OBJECTID	MODEL	MANUFACTURER
6	8	Mustang	1
6	9	Mustang	1
2	12	Accord	2
6	13	900s	4
6	14	9000	4
6	15	Prelude	2

5) automobile.retrieve revenue, customer, objectid, passengers

[RETRIEVE (TEMP=Automobile) (PASSENGERS, OBJECTID)
COMMON (OBJECTID, OBJECTID)
RETRIEVE (TEMP=Commercial) (CUSTOMER, REVENUE)]

PASSENGERS	OBJECTID	CUSTOMER	REVENUE
6	8	5	150
6	9	5	150
2	12	3	200
6	13	5	250
6	14	5	250
6	15	5	250

6) forgnauto.retrieve passengers, category

```
[RETRIEVE (TEMP=Forgnauto) (CATEGORY)
 COMMON (OBJECTID, OBJECTID)
 RETRIEVE (TEMP=Automobile) (PASSENGERS)]
```

CATEGORY	PASSENGERS
Compact	2
Compact	6
Compact	6
Sports	6

7) forgnauto.retrieve manufacturer, model, objectid, category

```
[RETRIEVE (TEMP=Forgnauto) (CATEGORY, OBJECTID)
 COMMON (OBJECTID, OBJECTID)
 RETRIEVE (TEMP=Vehicle) (MODEL, MANUFACTURER)]
```

CATEGORY	OBJECTID	MODEL	MANUFACTURER
Compact	12	Accord	2
Compact	13	900s	4
Compact	14	9000	4
Sports	15	Prelude	2

8) forgnauto.retrieve revenue, customer, objectid, category

```
[RETRIEVE (TEMP=Forgnauto) (CATEGORY, OBJECTID)
 COMMON (OBJECTID, OBJECTID)
 RETRIEVE (TEMP=Commercial) (CUSTOMER, REVENUE)]
```

CATEGORY	OBJECTID	CUSTOMER	REVENUE
Compact	12	3	200
Compact	13	5	250
Compact	14	5	250
Sports	15	5	250

9) commercial.retrieve revenue, customer, objectid

[RETRIEVE (TEMP=Commercial) (OBJECTID, CUSTOMER, REVENUE)]

OBJECTID	CUSTOMER	REVENUE
8	5	150
9	5	150
10	3	3
11	16	4
12	3	200
13	5	250
14	5	250
15	5	250

10) company.retrieve location, name, objectid

[RETRIEVE (TEMP=Company) (OBJECTID, NAME, LOCATION)]

OBJECTID	NAME	LOCATION
1	Ford	Newark
2	Chev	Detroit
3	Metro	Lansing
4	Cityline	Macon
5	National	Newyork
16	Honda	Tokyo
17	Saab	Gutenburgh

11) forgnco.retrieve location, name, objectid, country

[RETRIEVE (TEMP=Forgnco) (COUNTRY, OBJECTID)
COMMON (OBJECTID, OBJECTID)
RETRIEVE (TEMP=Company) (NAME, LOCATION)]

COUNTRY	OBJECTID	NAME	LOCATION
Japan	16	Honda	Tokyo
Sweden	17	Saab	Gutenburgh

12) vehicle.retrieve manufacturer.location, manufacturer.name, model

```
[RETRIEVE (TEMP=Vehicle) (MODEL)
 COMMON (MANUFACTURER, OBJECTID)
 RETRIEVE (TEMP=Company) (NAME, LOCATION)]
```

MODEL	NAME	LOCATION
Pinto	Ford	Newark
Camaro	Metro	Lansing
Mustang	Ford	Newark
Mustang	Ford	Newark
F100	Chev	Detroit
F150	Ford	Newark
Accord	Chev	Detroit
900s	Cityline	Macon
9000	Cityline	Macon
Prelude	Chev	Detroit

13) commercial.retrieve customer.location, customer.name, revenue

```
[RETRIEVE (TEMP=Commercial) (REVENUE)
 COMMON (CUSTOMER, OBJECTID)
 RETRIEVE (TEMP=Company) (NAME, LOCATION)]
```

REVENUE	NAME	LOCATION
150	National	Newyork
150	National	Newyork
3	Metro	Lansing
4	Honda	Tokyo
200	Metro	Lansing
250	National	Newyork
250	National	Newyork
250	National	Newyork

14) automobile.retrieve passengers, revenue if passengers > 4 and revenue < 200

```
[RETRIEVE ((TEMP=Automobile) and (PASSENGERS>4)) (PASSENGERS)
 COMMON (OBJECTID, OBJECTID)
 RETRIEVE ((TEMP=Commercial) and (REVENUE<200)) (REVENUE)]
```

PASSENGERS	REVENUE
6	150
6	150

15) automobile.retrieve passengers, revenue if passengers > 4 and revenue < 150

```
[RETRIEVE ((TEMP=Automobile) and (PASSENGERS>4)) (PASSENGERS)
 COMMON (OBJECTID, OBJECTID)
 RETRIEVE ((TEMP=Commercial) and (REVENUE<150)) (REVENUE)]
```

No such data is found.

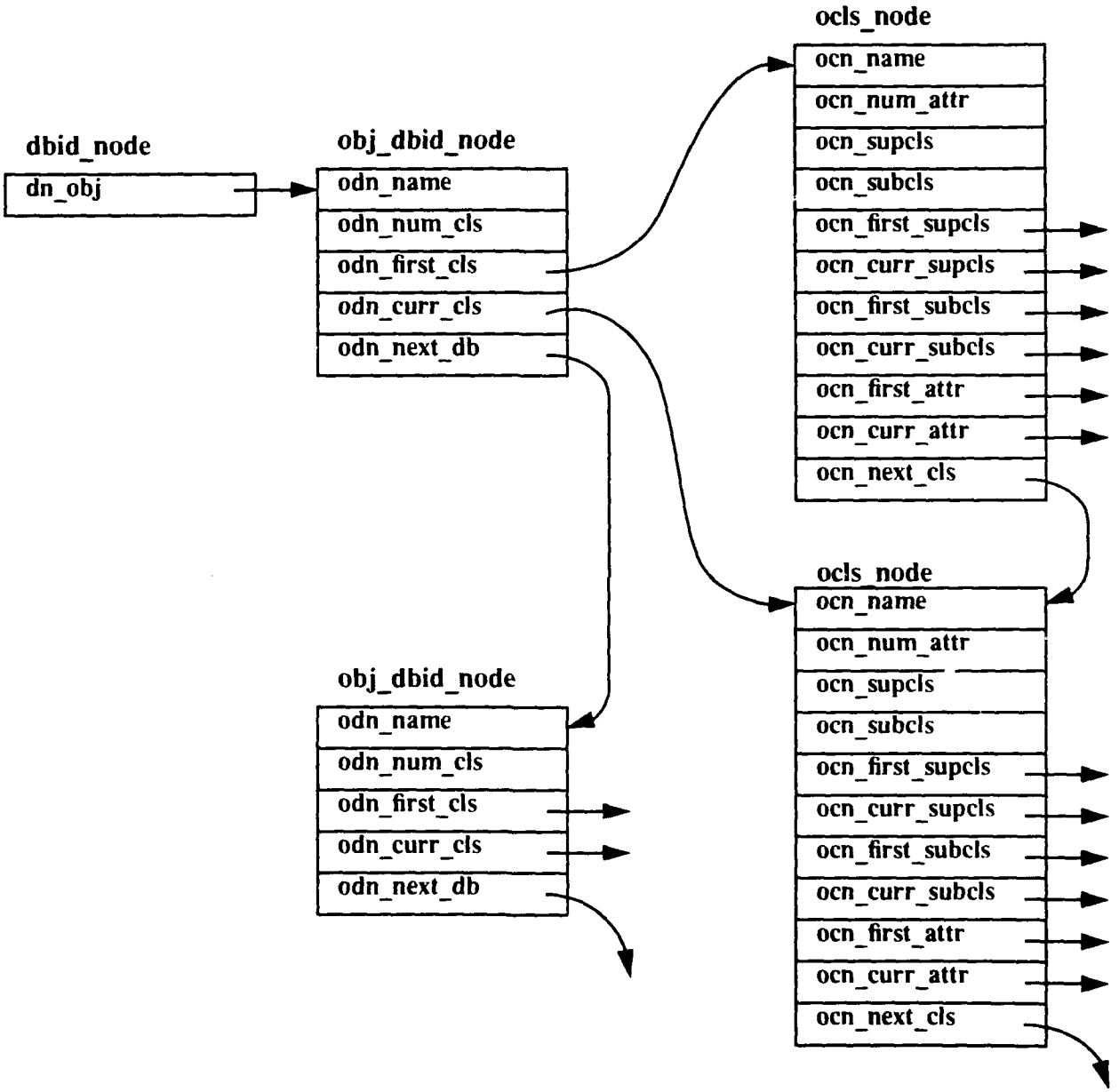
16) vehicle.retrieve manufacturer.name, model if manufacturer.location = 'Newark'

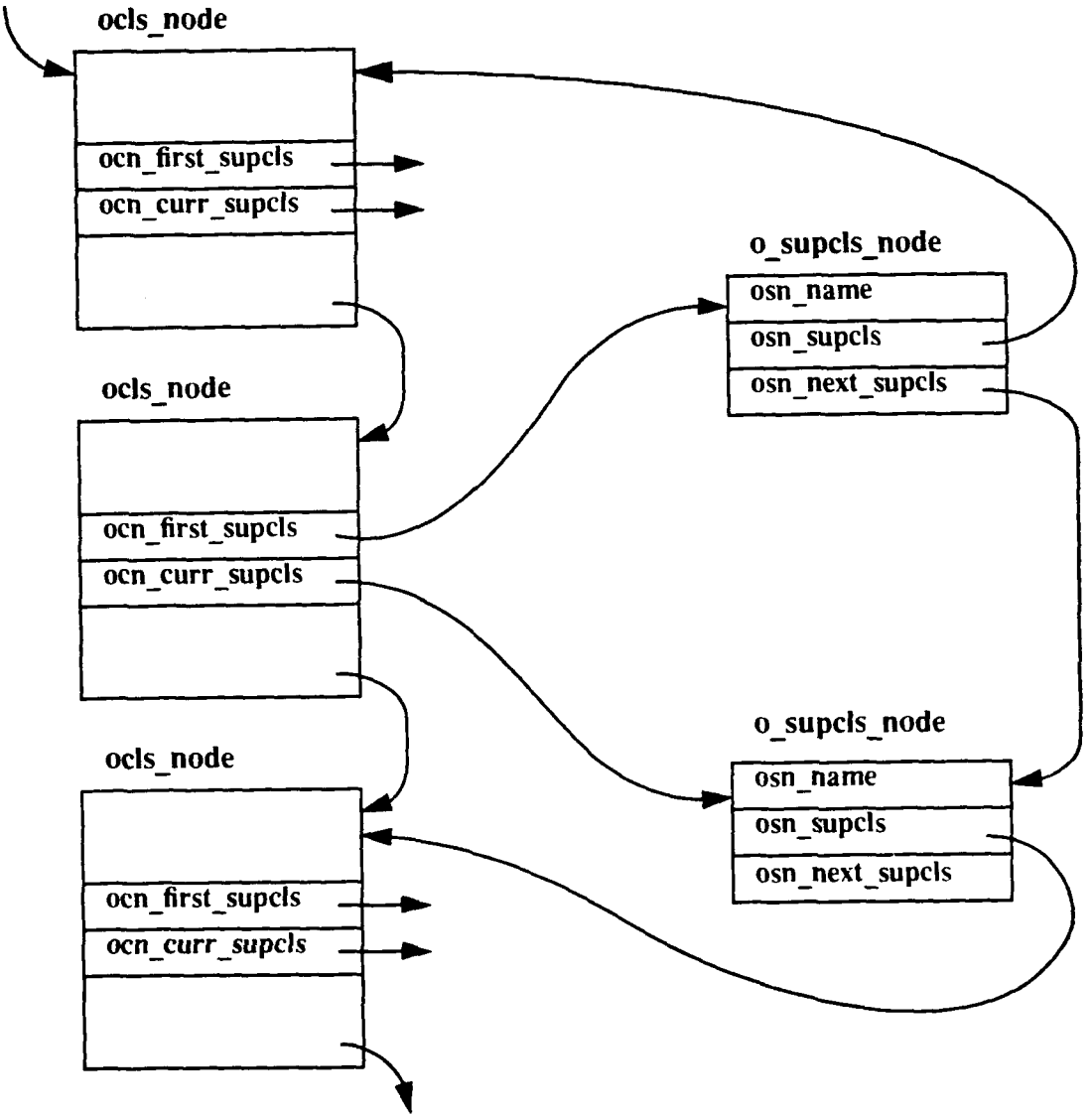
```
[RETRIEVE (TEMP=Vehicle) (MODEL)
 COMMON (MANUFACTURER, OBJECTID)
 RETRIEVE ((TEMP=Company) and (LOCATION='Newark')) (NAME)]
```

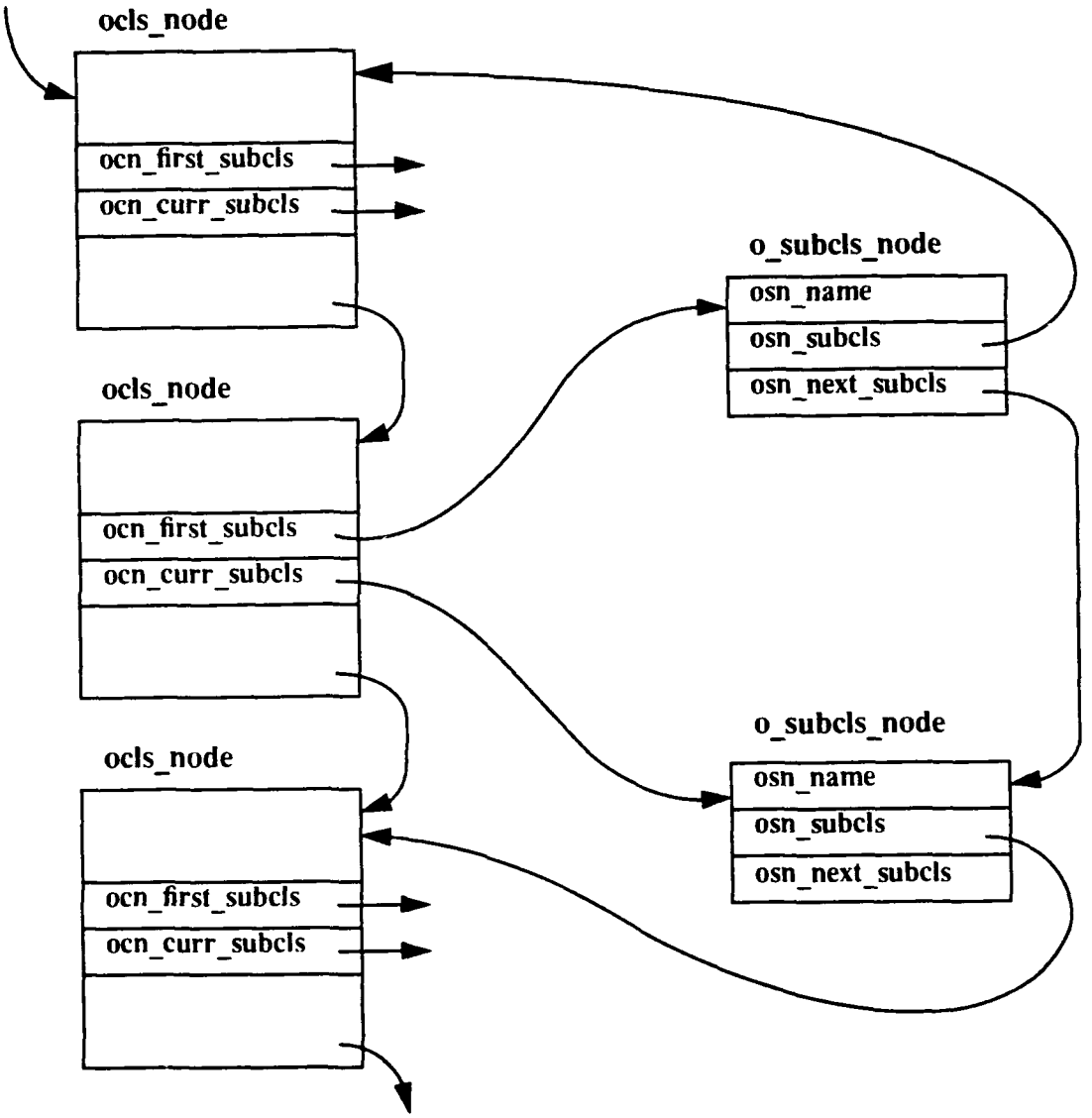
MODEL	NAME
Pinto	Ford
Mustang	Ford
Mustang	Ford
F150	Ford

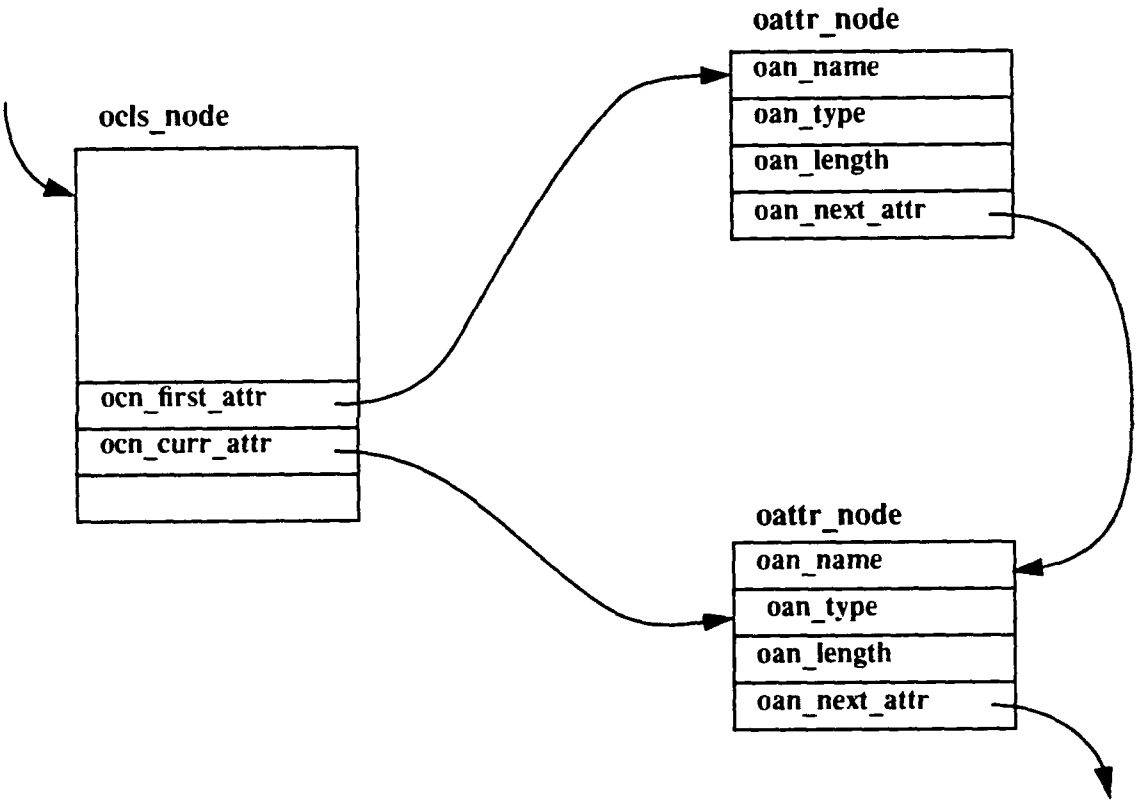
APPENDIX C

THE OBJECT-ORIENTED DATA STRUCTURES




ocls_node
ocn_first_supcls
ocn_curr_supcls
ocls_node
ocn_first_supcls
ocn_curr_supcls
ocls_node
ocn_first_supcls
ocn_curr_supcls
o_supcls_node
osn_name
osn_supcls
osn_next_supcls
o_supcls_node
osn_name
osn_supcls
osn_next_supcls


ocls_node
ocn_first_subcls
ocn_curr_subcls
o_subcls_node
osn_name
osn_subcls
osn_next_subcls
ocls_node
ocn_first_subcls
ocn_curr_subcls
o_subcls_node
osn_name
osn_subcls
osn_next_subcls
ocls_node
ocn_first_subcls
ocn_curr_subcls


oattr_node
oan_name
oan_type
oan_length
oan_next_attr
ocls_node
ocn_first_attr
ocn_curr_attr
oattr_node
oan_name
oan_type
oan_length
oan_next_attr

LIST OF REFERENCES

[Bour93] Bourgeois, Paul; *The Instrumentation of the Multimodel/Multilingual User Interface*, Masters Thesis, Naval Postgraduate School, Monterey, California, March 1993.

[Bert91] Bertino, Elisa; Martino, Lorenzo; "Object-Oriented Database Management Systems: Concepts and Issues," *IEEE Computer*, April 1991.

[Booc91] Booch, G., *Object Oriented Design*, The Benjamin/Cummings Publishing Company, Inc., 1991.

[Daws89] Dawson, J., "A Family of Models," *BYTE*, September 1989, pp. 277-286.

[Demu87] Demurjian, S., *The Multi-Lingual Database Systems - A Paradigm and Test-bed for the Investigation of Data-Model Transformations, Data-Language Translations and Data-Model Semantics*, Ph.D. Dissertation, Ohio State University, 1987.

[DemS87] Demujian, S.A. and Hsiao, D.K., "The Multi-Lingual Database System," *Proceedings of the 1987 3rd International Conference on Data Engineering, IEEE Computer Society Press*, February 1987.

[Demu89] Demujian, S.A. and Hsiao, D.k., "The Multi-Model Database System," *Proceedings of the International Phoenix Conference on Computers and Communications*, Phoenic, Arizona, March, 1989.

[Elma89] Elmarsi, R., Navathe, S., *Fundamentals of Database Systems*, The Benjamin/ Cummings Publishing Company, Inc., 1989.

[Hsia83] Hsiao, David K., Kerr, D.S., Orooji, A., Shi, Z. and Strawser, P., "The Implementation of a Multi-backend Database System (MDBS): Part I - An Exercise in Database System Engineering," (Chapters 10 and 12) *Advanced Database Machine Architecture*, Prentice-Hall, 1983, pp. 300-326, 327-385.

[Hsia89] Hsiao, David K. and Kamel, M.N., "Heterogenous Databases: Proliferations, Issues and Solutions," *IEEE Transactions on Knowledge and Data Engineering*, KDE-1, 1, 1989.

[Hsia91] Hsiao, David K. and Kamel, M.N., " The Multimodel and Multilingual Approach to Interoperability of Multidatabase Systems," *International Conference on Interoperability of Multidatabase Systems*, Kyoto, Japan, Aperil 1991.

[HsDK92] Hsiao, David K., "Federated Databases and Systems - Part I: A Tutorial on its Data Sharing," *VLDB* Journal, 1, 1992, pp. 127-179

[Hsia92] Hsiao, David K., " The Object-Oriented Database Management - A Tutorial on its Fundamentals," *Proceedings of the Second Far-East Workshop on Future Database Systems*, Kyoto, Japan, April 1992.

[HsiD92] Hsiao, David K., "A Parallel, Scalable, Microprocessor-Based Database Computer for Performance Gains and Capacity Growth," *IEEE MICRO*, December 1992, pp. 44-60.

[Hugh91] Hughes, John, *Object-Oriented Databases*, Prentice Hall, 1991.

[John93] Johnston, Richard, *The Relational-to-Object-Oriented Cross-Model Accessing Capability in a Multimodel and Multilingual Database System*, Masters Thesis, Naval Postgraduate, Monterey, California, March 93.

[Kell89] Kelley, Al; Pohl, Ira, *A Book on C*, The Benjamin Cummings Publishing Company, Inc., 1984.

[Kim89] Kim, W., Lochovsky, F., (Editors), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, 1989.

[Kim90] Kim, Won, "Object-Oriented Databases: Definition and Research Directions," *IEEE Transactions on Knowledge and Data Engineering*, Vol 2, NO. 3, pp 327-341 September 1990.

[Lipp92] Lippman, Stanley B., *C++ Primer*, 2nd Edition, Addison-Wesley Publishing Company, 1991.

[Roll84] Rollins, R., *Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate, School, Monterey, California, June 1984.

[Schl90] Schlageter Rainer U., "Object-Oriented Database Systems: Concepts and Perspectives," *Lecture Notes in Computer Science #466*, pp. 154-197, 1990.

[Tsud91] Tsuda, K., Yamamota, K., Harakawa, M. and Ichikawa, T., "MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 4, December 1991

[Wort85] Wortherly, C., *The Design and Analysis of a Network Interface for the Multi-Lingual Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[Weis84] Weishar, D., *The Design and Analysis of a Complete Hierarchical Interface for the Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, June 1984.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943	2
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 37 CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Dr. David K. Hsiao Code CS/Hs Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
Doris Mlezco Code 9033 Naval Pacific Missile Test Center Point Mugu, CA. 93042-5001	1
Officer in Charge Patrol Squadron Special Projects Unit One Naval Air Station Brunswick, ME. 04011-5000	1
Deniz Kuvvetleri Komutanligi Personel Daire Baskanligi Bakanliklar, Ankara / TURKEY	1

Golcuk Tersanesi Komutanligi
Golcuk, Kocaeli / TURKEY 2

Deniz Harp Okulu Komutanligi
81704 Tuzla, Istanbul / TURKEY 2

Taskizak Tersanesi Komutanligi
Kasimpasa, Istanbul / TURKEY 2

LT John W. Moore
Patrol Squadron Special Projects Unit One
Naval Air Station
Brunswick, ME 04011-5000 2

LTjg Turgay Karlidere
Vicdaniye Mah. Sakarya Cad. No: 63/2
10030 Balikesir / TURKEY 1